ERDC MSRC Major Shared Resource Center

ERDC MSRC/PET TR/00-15

Modeling of HPC Platforms and Performance Tuning of DoD Applications

by

Wenheng Liu
Neungsoo Park
Jinwoo Suh
Santosh Narayanan
Viktor K. Prasanna

26 May 2000

06h0042000

Work funded by the DoD High Performance Computing Modernization Program ERDC Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
CSC Nichols

Modeling of HPC Platforms and Performance Tuning of DoD Applications

Wenheng Liu, Neungsoo Park, Jinwoo Suh, Santosh Narayanan, Viktor K. Prasanna
Department of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089-2562
URL: http://ceng.usc.edu/˜prasanna.html

May 26, 2000

Abstract

Memory access cost is an important consideration in performing data intensive applications. The widening gap between the performance of processor and memory magnifies the importance of data placement during computation. In this report, we describe techniques to improve memory system performance. These techniques involve data placement and program control tuning. The application and algorithm designer can apply them to reduce the total memory access cost during computation. We demonstrate an approach by considering matrix transposition, multiplication, and mesh generation of generic finite element methods. We use our Integrated Memory Hierarchy (IMH) model to design data layouts for efficient matrix operations. Our proposed work provides a uniform methodology across multiple HPC platforms for the development of a suite of the kernel codes (e.g., matrix transposition and matrix multiplication) commonly used by DoD applications.

We present an efficient algorithm for out-of-core matrix transpose for transposing large-scale matrices. Our algorithm improves execution time by reducing both the number of I/O operations and the index computation time. I/O operations are reduced by using efficient data layout on disk and balancing the number of reads and writes.

For ”standard” matrix multiplication, we have developed a novel data layout that reduces cache pollution, data cache misses and TLB (Translation Look-Aside Buffer) misses. Prior to computation, we reorganize the matrix data layout by transposing and partitioning into blocks, each equal to the virtual page size. This proposed approach avoids cache pollution, conflict cache misses, and TLB misses.

The third problem is implementation of appropriate mesh generation for a suite of Finite Element Methods (FEMs). We apply an algorithmic technique of separating rows of a data matrix based on their access patterns. Applied in conjunction with blocking, our approach minimizes memory requirements and optimizes cache performance.

We have implemented our proposed approaches on UltraSPARC II, Alpha 21264, Cray T3E and Pentium III based machines. Experimental results show our approaches to be effective in improving overall performance.

Contents

1 Introduction

HPC platforms are being employed to perform a wide variety of data-intensive applications such as scientific computations, media processing, automatic target recognition systems among others [5, 8, 12, 17, 18, 20, 21, 23, 24]. In such data-intensive applications, the data is stored in different levels of a memory hierarchy with different data access costs. Although HPC platforms are already able to provide large computational power, the memory access costs have not improved accordingly. Thus memory access can become the bottleneck in performing such data-intensive applications on HPC platforms.

In this report, we focus on algorithmic techniques for efficient memory access. We show the effectiveness of our approach by showing improved performance for two kernel operations, matrix transposition and matrix multiplication, and the mesh generation operation of generic finite element methods on state-of-the-art machines. The efficiency of such operations depends heavily on the data access cost. Traditionally, matrices are stored in either column or row major order (data layout) in the memory. Mismatch between the data layout and the data access pattern can lead to severe performance degradation due to cache misses, cache pollution, DRAM page faults, and TLB misses. In this report, algorithmic techniques such as cache conscious data layout techniques are proposed for performance tuning of these matrix operations. Our design of efficient data layouts for matrix operations is based on our previously proposed Integrated Memory Hierarchy (IMH) model. The IMH model is used to predict the performance of DoD applications. Our proposed work provides a uniform methodology across multiple HPC platforms for performance optimization of a suite of the kernel codes (e.g., matrix transposition and matrix multiplication) used by many DoD applications.

Efficient transposition of large-scale matrices has been widely studied. These efforts have focused on reducing the number of I/O operations. However, in the state-of-the-art architectures, data transfer time and index computation time are also significant components of the overall time. In this research effort, we developed an algorithm that considers all these costs and reduces the overall execution time.

Our algorithm reduces the number of I/O operations significantly by using two techniques: (1) writing the data onto disk in predefined patterns and (2) balancing the numbers of read and write operations to disk. The reduction of the number of I/O operations has a significant impact on overall time for I/O, which is measured by elapsed wallclock time. The reason for this is that the startup time for an I/O operation is several orders of magnitude greater than the time for transferring an actual data byte, in state-of-the-art disk systems. The idea behind balancing the I/O operations is to reduce the number of write operations at the expense of an increased number of read operations, so that the total number of I/O operations is reduced compared with the state-of-the-art. The index computation time, which is an expensive operation involving two divisions and a multiplication, is eliminated by partitioning the memory into two buffers. The expensive in-processor permutation is replaced by data collection operations. Our algorithm is analyzed using the well-known Linear

Model and the Parallel Disk Model. The experimental results on a Sun Enterprise and a DEC Alpha show that our algorithm reduces the execution time by about 50%, compared with the best known algorithms in the literature.

In addition, a novel data layout is proposed to reduce cache pollution and data cache and Translation Look-Aside Buffer (TLB) misses in performing the ”standard” matrix multiplication. We describe the costs of data access in the memory hierarchy and the issues in data layout design. Then, we describe our efficient data layout for the standard matrix multiplication operation and compare the performance to three other approaches. Experimental results on UltraSPARC II, Alpha 21264, Cray T3E and Pentium III based machines show improved performance.

The finite element method (FEM) is widely used for analysis and simulation of a large variety of scientific problems. The first step of a typical FEM is to generate an appropriate mesh for the problem domain. In grand challenge applications, a typical FEM has 10,000 to 1,000,000 elements. Such applications require a large amount of memory accesses. In this report, we describe a data layout for the mesh generation of an FEM using serendipity elements. In our technique rows of a data matrix are clustered based on their access patterns. This technique can avoid the cache pollution problem. Then, blocking and block layout are used to further reduce cache misses and TLB misses. This data layout results in efficient data access and minimizes the memory space allocated for the corresponding mesh.

The rest of the report is organized as follows. Section 2 gives a description of the model of HPC platforms developed as part of this research effort. Section 3 describes our novel algorithm for matrix transposition. Section 4 presents our techniques for matrix multiplication. Section 5 discusses our research on the use of efficient data layouts for FEM applications and Section 6 concludes the report.

2 Our Model of HPC Platforms

In this section, we describe previous memory models and present our Integrated Memory Heirarchy model for HPC patforms.

2.1 Previous Models

The Parallel Memory Hierarchy (PMH) model [3] and the Two-Level Memory Model have been proposed for HPC platforms. These are briefly described here.

2.1.1 Parallel Memory Hierarchy Model

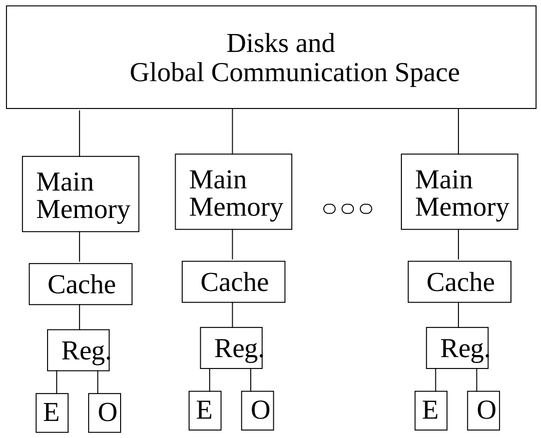


Figure 1: PMH model of the IBM SP1. Boxes labeled E (for EVEN) and O (ODD) are functional units that model the two-stage floating-point pipeline.

In this model, the interprocessor communication cost and the memory hierarchy are considered. A parallel computer is modeled as a tree of modules. Each non-leaf node represents a memory module such as disk, main memory, cache, and register. A leaf node represents a computing element such as a functional unit in a CPU. The PMH model of the IBM SP1 is shown in Figure 1. Each child connects to its parent by a unique channel. Modules hold data. Data in a module are partitioned into blocks. A block is the unit of transfer on the channel connecting a module to its parent. The model has four parameters for each module m: s_m is the number of bytes per block of m, n_m is the number of blocks in m, c_m is the number of children of m, and t_m is the number of cycles to transfer a block between m and its parent.

This model considers the interprocessor communication and the secondary memory access. However, the model can not represent the hard disk system if the hard disk is distributed among processor nodes. Examples of the architectures using this model are shown in Figure 2. In Figure 2, (b) shows a shared disk system. The processors are interconnected by a high-bandwidth network. Figure 2, (a) shows the distributed disk system where the disks are interconnected by a low-bandwidth network. The model inherently assumes that the interprocessor communication is performed through the disk. Hence, the model can not represent a distributed disk system in which the processors are interconnected by a high-bandwidth network.

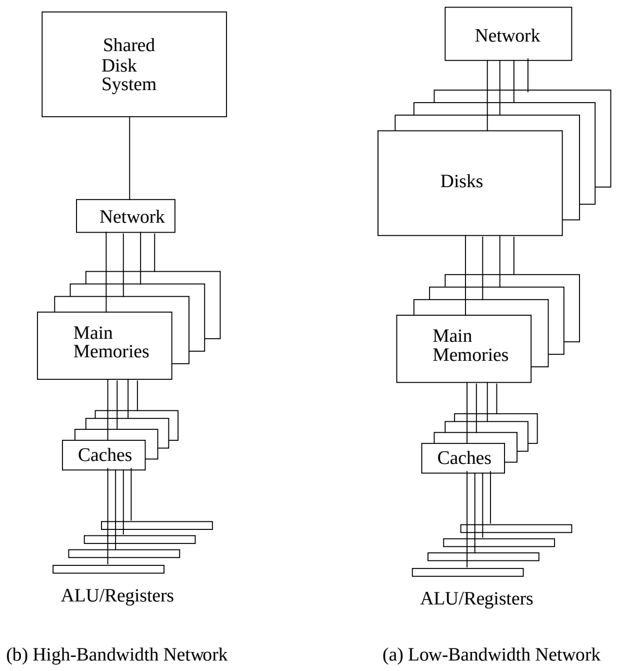


Figure 2: PMH model of parallel systems

Another drawback of this model is high complexity. Since every memory module including cache and registers is modeled using four parameters, the resulting model is too complicated. The more parameters a model has, the more difficult it is to design and optimize algorithms.

2.1.2 Two-Level Memory Model

This model has been proposed for the development of parallel input/output algorithms. The underlying architecture is shown in Figure 3. In this model, the number of input/output operations is used to estimate the communication cost.

The data transfer time to or from the disk is ignored because the seek time in a disk access operation is much larger than the data transfer time for small data sizes. However, the data transfer time is an important factor when the data size is large. The disadvantage of this model is that the communication time between processors is completely ignored because communication time between processors is insignificant compared with the disk access time. However, it is an important factor if the number of communication operations or the amount of data transferred is large.

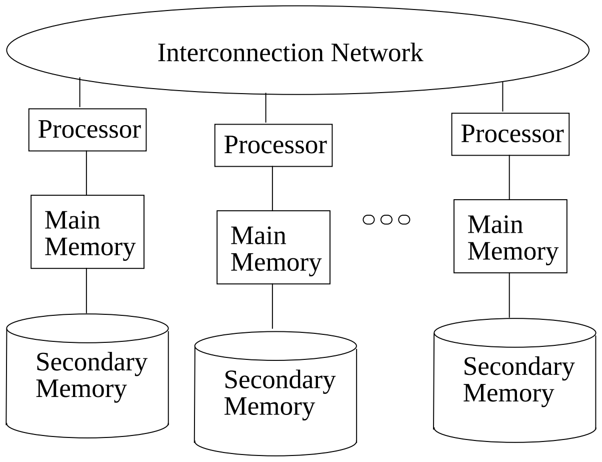


Figure 3: HPC architecture used in two-level memory model

2.2 Integrated Memory Hierarchy Model

Our HPC platform model considers three main costs: processor-processor, processor-memory, and memory-disk costs. The processor-memory and memory-disk costs involve only communication cost. The processor-processor cost consists of computation and communication costs between processor and memory.

2.2.1 Processor-Memory

The results of the Integer READ operation experiment is shown in Figure 5 and Figure 6 to illustrate our approach to modeling the processor-memory communication cost. The results of this out-of-cache operation shows the importance of two key parameters: the number of elements (N) and the stride (S). In Figure 4, the execution time of reading arrays with various number of elements is shown. As can be seen in the figure, the total execution time increases linearly as the number of elements is increased.

In Figure 5 and 6, the stride is varied for each of the arrays (with various number of elements) shown in Figure 4. As the stride is increased, the total execution time increases linearly for each array, and peaks out towards the end. The peak point occurs when the stride is large enough that each access incurs a cache miss. This phenomenon can be clearly seen in Figure 6. In this graph, the difference in execution time for each consecutive array size from Figure 5 is shown. This graph, in essence, shows the cost of executing 512 additional elements starting from some base size.

The cost of communication increases linearly as the number of elements N increases for the basis stride = 1. This cost, T, can be modeled using the following linear equation

$$T = T_n \times N_n \tag{1}$$

where T_n is the time for transferring a byte of data between the processor and memory and N_n is

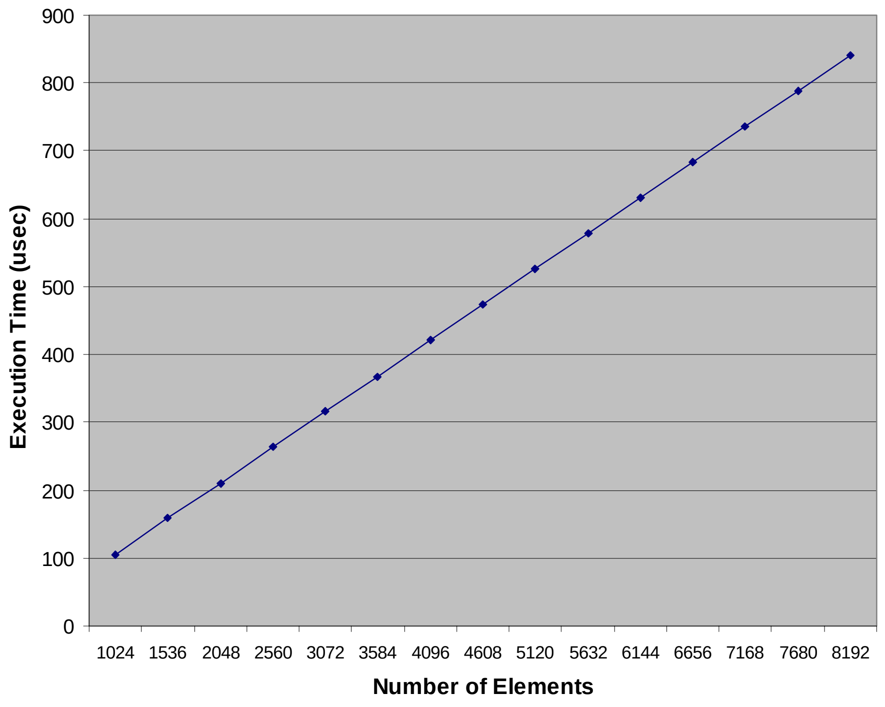


Figure 4: Read Integer using various number of elements

the number of bytes transferred between the processor and memory.

As stride S increases, the total size of the data increases linearly until S is equal to the cache line size. Also, this can be modeled using a linear equation

$$T = T_c \times N_c \tag{2}$$

where T_c is the time to bring a cache line to the cache, and N_c is the number of cache lines transferred to cache.

Table 1: Parameter Values for Processor-Memory Cost

Platform	T_n	T_c	T_e
SP	120 nsec	135 nsec	100 nsec

2.2.2 Processor-Processor

Our model of processor-processor communication uses results of permutation communication, since it can be considered as a general communication pattern. For example, an all-to-all communication can be implemented using many steps of permutation communication.

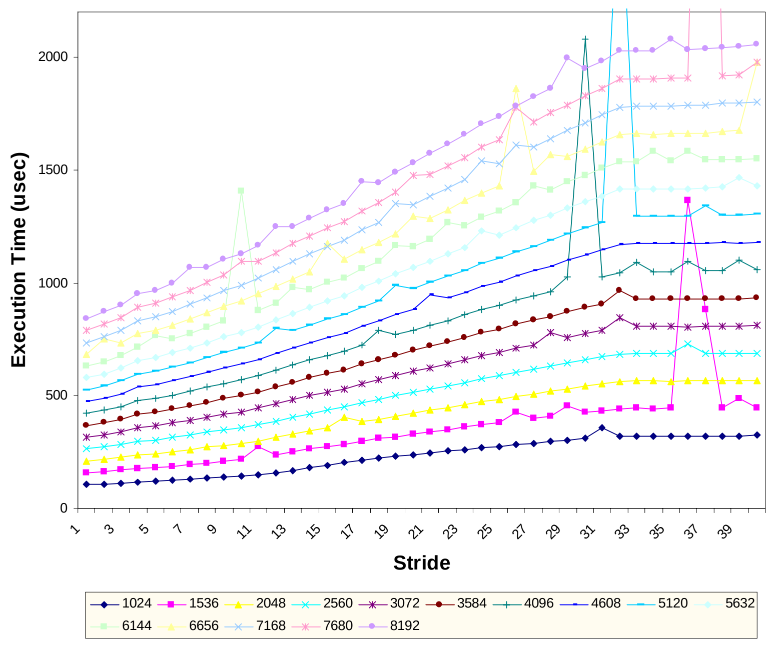


Figure 5: Processor-Memory communication: Read Integer

Figure 7 shows the permutation communication time as a function of message size. In Figure 8, the lower-left corner is enlarged. Figure 7 shows that the communication cost is proportional to message size. However, when the message size is small (See Figure 8), there is a relatively large communication startup cost. Based on these observations, the communication time between two processors can be modeled using a linear function of the message size, m, as follows:

$$\text{Communication time between a pair of processors} = T_s + m\tau_d \tag{3}$$

where T_s = startup time, and $\tau_d = 1/\text{bandwidth}$ = data transfer time per byte per processor. The T_s and τ_d are obtained using our permutation communication benchmark experimental results. The parameters for the SP and the T3E are shown in Table 2.

Table 2: Startup Time and Bandwidth

Platform	Startup Time (μsec)	Bandwidth (MB/sec/processor)
SP	54	50
T3E	29	100
O2K	85	15

Using equation (3), the time to perform more complex communication patterns can also be

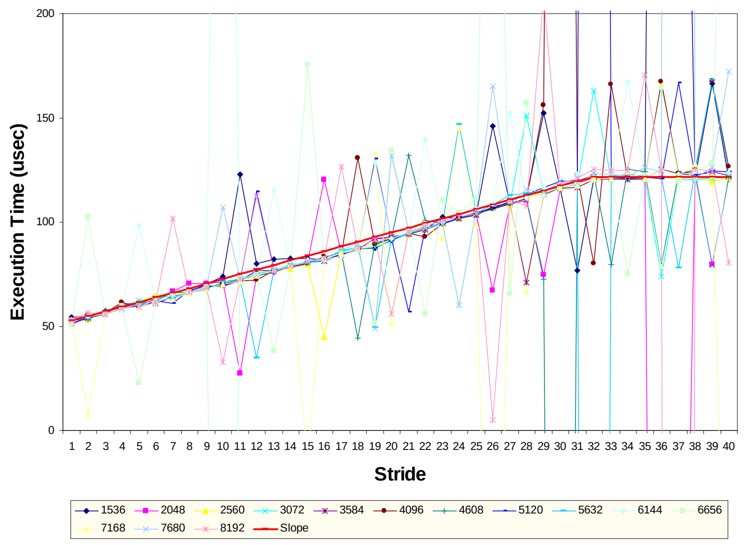


Figure 6: 512 Difference Graph: Read Integer

modeled. In these cases, we found that the startup time does not a show large variation for different communication patterns. However, the communication time depends on the total data size and the number of processors. Thus, when a communication pattern consists of j steps, the total message size is $\sum_{i=1}^{j} m_i$, and the total data transfer time is $\sum_{i=1}^{j} m_i \times \tau_d$. Therefore, the total communication time for a communication pattern is

$$\begin{aligned} \text{T}_{comm} &= \text{Startup time} + \text{Total data transfer time} \\ &= T_s + \sum_{i=1}^{j} m_i \tau_d \end{aligned}$$

where T_s = startup time, τ_d = data transfer time per byte, and j = the number of communication steps in the communication pattern.

With this equation, the communication time for various communication patterns can be estimated as follows:

- Permutation time = $T_s + m\tau_d$
- Pingpong time = $T_s + m\tau_d$
- Scatter time = $T_s + (P-1)m\tau_d/2$
- Broadcast time = $T_s + (\log P)m\tau_d/2$

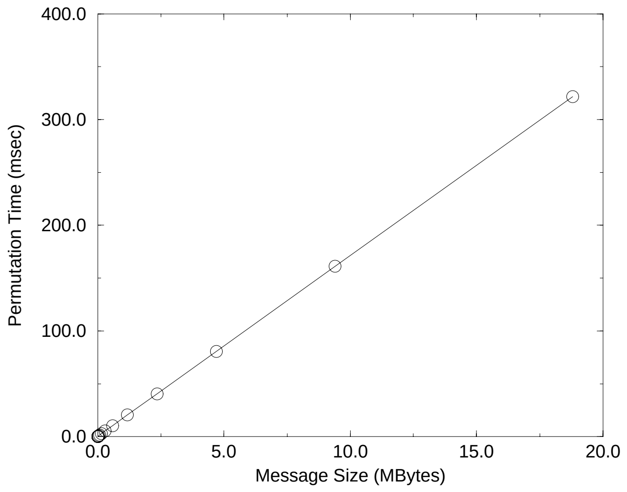


Figure 7: Permutation communication results on the SP

where m is the size of the message that is transferred to each destination processor, and P is the total number of processors involved in the communication.

To validate our model, we compared the estimated communication time and the actual communication time for each of our benchmarks. The following data block sizes were used: each communication is:

- For permutation and pingpong: 16 MB,
- For scatter among 8 processors: 16 MB on the root processor. 2 MB sent to each destination processor,
- For scatter among 16 processors: 16 MB on the root processor and 1 MB for the destination processors,
- For broadcast among 8 processors: 2 MB, and
- For broadcast among 16 processors: 1 MB.

In estimating the pingpong communication time, we used the fact that the architectures support "pipelined communication" discussed later in this report.

The results show that the model can accurately predict the communication time on the SP. On the T3E, the maximum error was about 30%. The error range can be further reduced by adjusting the parameters.

2.2.3 Memory-Disk

The read and write operation times are shown in Figure 9, Figure 10, and Figure 11 for the SP, the T3E, and O2K, respectively. The overall graph shows that the read and write times are proportional to the message size.

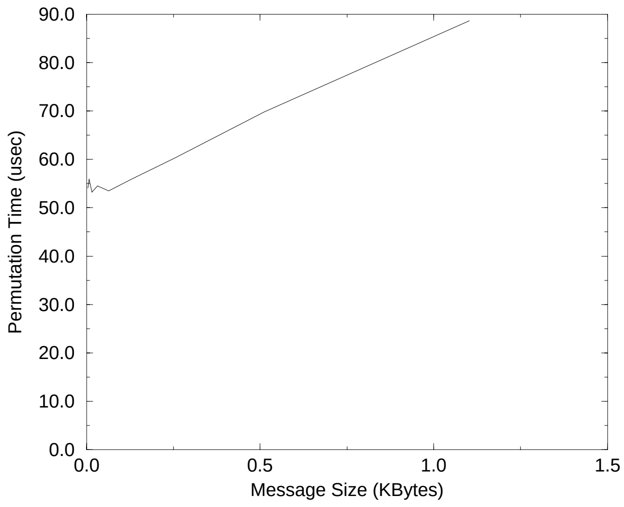


Figure 8: Permutation communication results on the SP

Table 3: Predicted Time and Actual Time (msec)

Communication operation	Actual	Predicted	Error
Permutation	320	320	0%
Pingpong	350	320	9.4%
Scatter on 8 proc.	160	140	14%
Scatter on 16 proc.	170	150	13%
Broadcast on 8 proc.	63	60	5%
Broadcast on 16 proc.	38	40	5%

The spikes at message size = 1.5 and 3.5 are due to random operating system behavior. We performed our experiments over many iterations and found that the spikes are random, i.e., there was no regular pattern nor consistency in the appearance of these spikes. From this, we conclude that the spikes are not related to any parameter nor characteristic of the underlying hardware platform. We reason that this is probably due to the operating system behavior and interactions with other jobs on the system.

The disk operations can be modeled using a linear equation as a function of the data size. However, the write operation takes more time than the read operation because the write operation needs to perform a read before the data is written to the disk, if the page containing the data is not in the memory. Thus, the read and the write operations are modeled using different parameters. Also, there is a large startup cost in the msec ranges. Even though recent technological advances have significantly improved the performance of disk, the startup cost is still large compared with data transfer time per byte. Hence, our model incorporates the startup cost. Therefore, the disk operation time can be modeled using a linear equation for the read and write operations. However, since the parameters for read and write operations are different, we use a separate linear equation for each

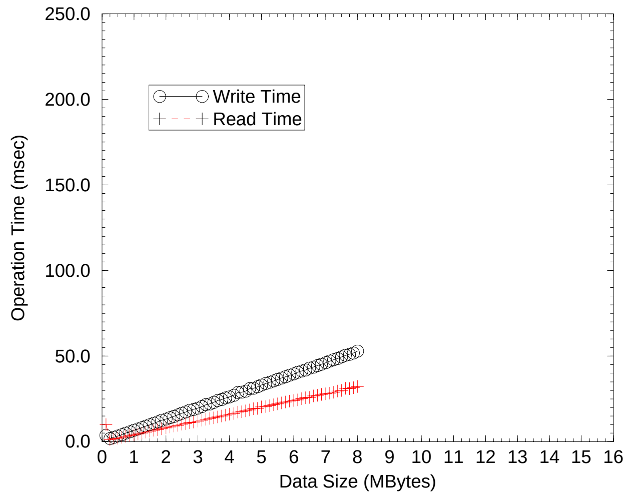


Figure 9: Disk operation results on the SP

operation.

$$
\begin{aligned}
\text{Disk operation time} &= \text{disk read time + disk write time} \\
&= (T_r + m_r\tau_r) + (T_w + m_w\tau_w)
\end{aligned}
$$

where T_r is startup time for the read operation, m_r is the read message size, τ_r is inverse of the read bandwidth, T_w is startup time for the write operation, m_w is the write message size, and τ_w is the inverse of the write bandwidth. The parameters obtained using our benchmark suites are summarized in Table 4.

Table 4: Memory-Disk Communication Time

Operation	Platform	Startup Time (msec)	Bandwidth (MB/sec)
Write	SP	1.0	155
	T3E	3.5	149
	O2K	2.0	70
Read	SP	1.0	255
	T3E	1.0	266
	O2K	1.5	90

2.2.4 Integrated Memory Hierarchy Model

The complete model is obtained by integrating the models for each communication. An overview of our model is shown in Figure 12. The complete model can be written as follows:

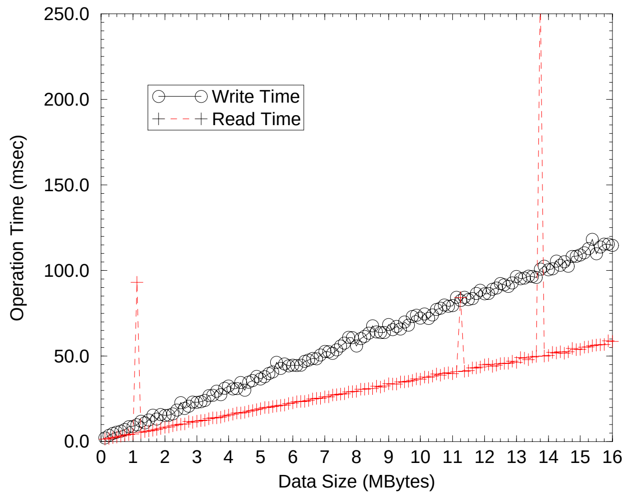


Figure 10: Disk operation results on the T3E

$$\begin{aligned}
\text{Execution time} &= \text{processor-memory execution time} \\
&\quad + \text{processor-processor communication time} \\
&\quad + \text{memory-disk communication time} \\
&= T_n \times N_n + T_c \times N_c + T_e \times S \\
&\quad + T_s + m\tau_d \\
&\quad + T_r + m_r\tau_r + T_w + m_w\tau_w
\end{aligned}$$

where T_n = the data transfer time between the processor and memory per byte,
N_n = the number of data elements that are transferred to cache,
T_c = the time to bring a cache line to the cache,
N_c = the number of cache lines that are transferred to the cache,
T_e = a constant to compensate for the difference in the slopes,
S = stride in which data is accessed,
T_s = startup time between processors,
m = size of the message transferred between processors,
$\tau_d = 1/\text{bandwidth}$,
T_r = startup time for the read operation,
m_r = the read message size,
τ_r = inverse of the read bandwidth,
T_w = startup time for the write operation,
m_w = the write message size, and
τ_w = the inverse of the write bandwidth.

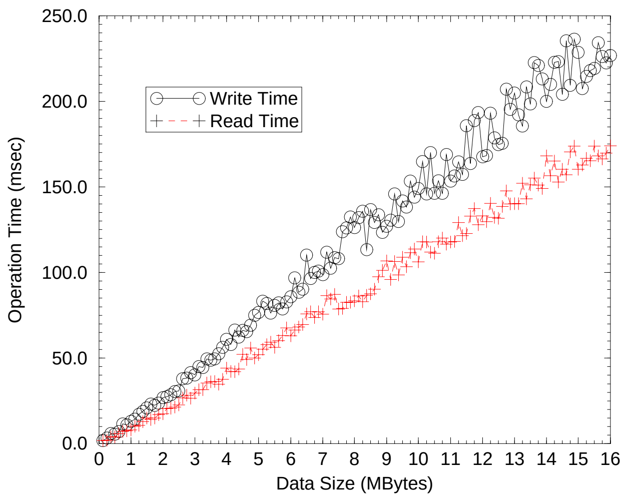
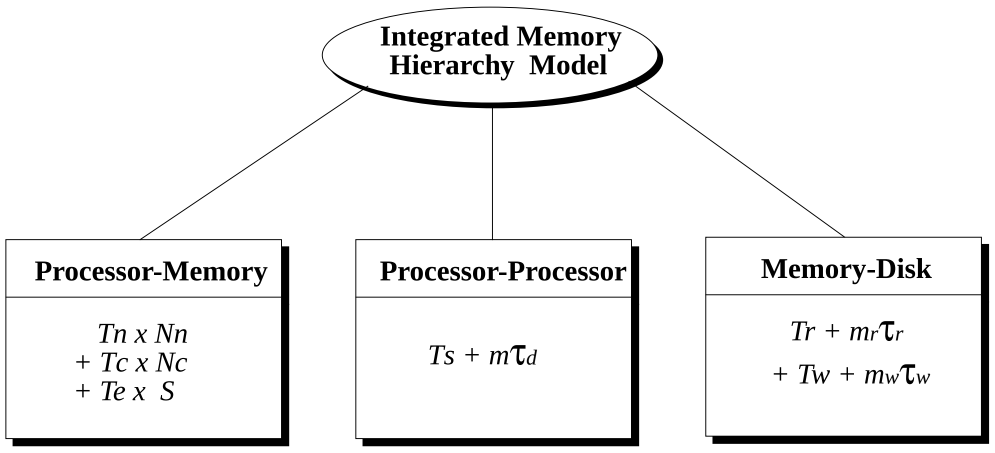


Figure 11: Disk operation results on the O2K



Integrated Memory Hierarchy Model
Processor-Memory
$Tn \times Nn + Tc \times Nc + Te \times S$
Processor-Processor
$Ts + m\tau_d$
Memory-Disk
$Tr + m_r\tau_r + Tw + m_w\tau_w$

Figure 12: An overview of the model of HPC platforms

2.3 Significance and Use of Our Model

Our model of HPC platforms is:

- As an integrated model, it consists of three main costs in peforming computation on HPC platforms: processor-memory, processor-processor, and memory-disk. The computation cost is included in the processor-memory cost. Thus, in our model, the costs for computation and communication among various HPC components are considered.
- As a simple model, an HPC platform consists of a number of components, each having a large number of parameters. To model HPC platforms with very high accuracy, a large number of

parameters need to be included in the model. Such a model becomes too complex to be of value to end-users. Therefore, we first identified the three main costs. These costs are modeled using simple equations providing users a simple view of HPC platforms. Also, we avoided discontinuous functions to avoid complex calculations. These efforts simplified the model for users.

- There is a trade-off between accuracy and simplicity. In our model, we sacrificed some amount of accuracy for simplicity. However, we obtained a model accurate enough for the design and analysis of algorithms on HPC systems. An example is shown in the next section.

- Our model is useful for design and analysis of algorithms on HPC platforms. Design and analysis of algorithms require understanding of HPC platforms on which the algorithm is used. Our model provides a simple and fairly accurate view of the HPC platforms. With the model, users can predict performance of their code. The users can optimize the code before the actual run. Also, after a test run, users can easily analyze the execution time using our model. Thus, the users can save time and effort in designing and analyzing algorithms on HPC platforms.

3 Matrix Transpose

In this section, we describe the general disk models, previous algorithms of matrix transpose, and details of our proposed algorithm to perform matrix transpose. Then, we show the performance improvement achieved by using our algorithm compared with the previous ones.

3.1 Disk Models

State-of-the-art disk systems employ sophisticated hardware and perform several optimizations to reduce the I/O time. For example, many of these systems employ a disk buffer, a library buffer, and a controller, and perform access reordering. Each of the above system features needs several parameters to describe its behavior and such a model will be too complex to be useful.

Two models of disk systems that capture the key characteristics of such systems have been widely used in the literature. One of them is the Parallel Disk Model (PDM) [25] (see Figure 14). It models the low level behavior of disk systems using several parameters: block size (B) which is the size of data that is transferred between disk and memory in one I/O operation, number of disks (D), memory size (M), number of processors (P), and amount of data transferred (m). This model has been used to study RAID systems. The total time for data transfer between disk and memory is proportional to the number of blocks transferred and is inversely proportional to the number of disks on which the data resides. Thus, the cost can be represented as $\lceil m/(DB) \rceil \times T_b$, where T_b is the time to transfer a block of data between memory and disk.

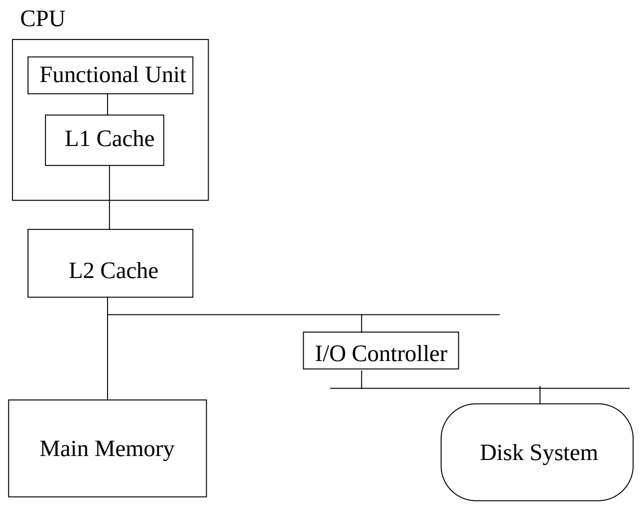


Figure 13: Typical Disk system

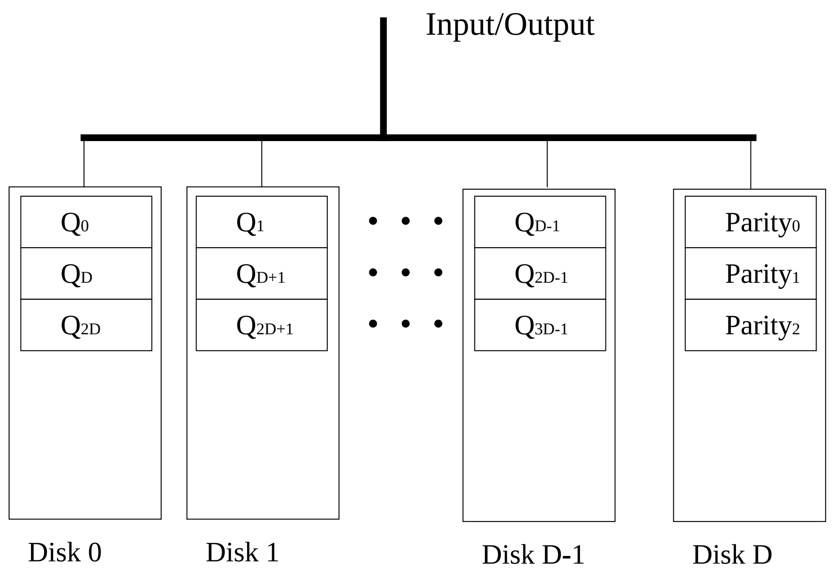


Figure 14: Typical RAID system (RAID 3)

```
1  for $s$ = 0 to $\lg_{M/B} \min(B, N/B)$ - 1 // for each stage
2      for $j$ = 0 to $N^2/M - 1$ // for each step
3          Read M/B blocks;
4          Perform permutation of data in memory;
5          Write M/B blocks;
```

Figure 15: Pseudo-code for Aggarwal et al.'s algorithm

In another model [15], two costs are considered: startup time and data transfer time. The startup time is a fixed time for setting up the data transfer between memory and disk. The rest of the cost is proportional to the amount of data transferred. Thus, it can be represented as $T_s + m\tau$, where T_s is the startup time, m is the data size, and τ is the time to transfer unit data. Typically, T_s is in the msec range, and τ is in the tens of nsec/byte range.

3.2 Previous Algorithms

In this section, for the sake of completeness, two well-known algorithms are briefly described. These two algorithms provide the best performance among many other algorithms. The algorithm in [1] has been designed using the Parallel Disk Model (PDM) and the algorithm in [15] has been designed using the Linear Model. In Section 4, our algorithm is compared with these algorithms.

3.2.1 Matrix Transpose

In the matrix transpose problem, an input matrix of size $N \times N$ initially resides on the disk. $N = \prod_{s=0}^{t-1} r_s$, where r_s is a positive integer. If N is a prime number, we can add dummy rows to make N nonprime. The input matrix is to be transposed and stored in the other array. The available memory size, M, is assumed to be smaller than the input matrix size. Throughout this paper, to illustrate the key ideas, we use square matrices. However, the algorithms can be easily extended to rectangular matrices as well using the technique in [15]. For the sake of simplicity, throughout this paper, we assume that all the ratios are integers.

3.2.2 Aggarwal's Algorithm

Aggarwal et al. [1] showed a lower bound on the number of I/O operations to perform matrix transpose. A pseudo code for the algorithm is shown in Figure 15. In this algorithm, as many blocks as the size of the available memory are read into memory. Then, the data is permuted and written onto the disk.

In this algorithm, r_s is restricted to be $\leq M/B, 0 \leq s \leq t-1$. In our algorithm, we relax this restriction by developing a technique to use a larger block size. Also, this algorithm does not consider index computation time. Index computation is needed to perform permutation of the data

```
1  for $s$ = 0 to $t$-1 // for each stage
2      for $j$ = 0 to $M/B - 1$ // for each step
3          Read $M$ amount of data;
4          Perform permutation of data in memory;
5          for $k$ = 0 to $r_s - 1$
6              Write $M/B$ amount of data;
```

Figure 16: Pseudo-code for Kaushik et al.'s algorithm

in memory.

3.2.3 Kaushik's Algorithm

In this algorithm [15], there are t stages, where $N = \prod_{s=0}^{t-1} r_s$. Each stage consists of N^2/M steps. In each step, M/N rows are read into memory and a permutation of the data is performed in the memory. Then, the data is written back to the disk in $r_s, 0 \leq s \leq t-1$, write operations. Thus, the number of read (write) operations in each step is 1 (r_s). A pseudo code for the algorithm is shown in Figure 16.

Although the number of I/O operations and the time to transfer data between memory and disk are considered, the total number of read and write operations are not optimized. Also, the index computation time is not considered.

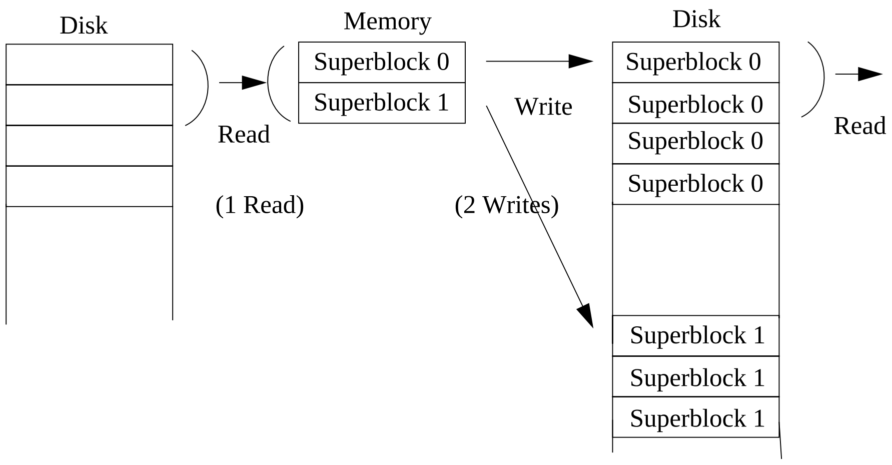
3.3 An Efficient Algorithm

We present an overview of our approach in Section 3.3.1. The subsequent sections provide all the details of our approach and analyses using the PDM and LM.

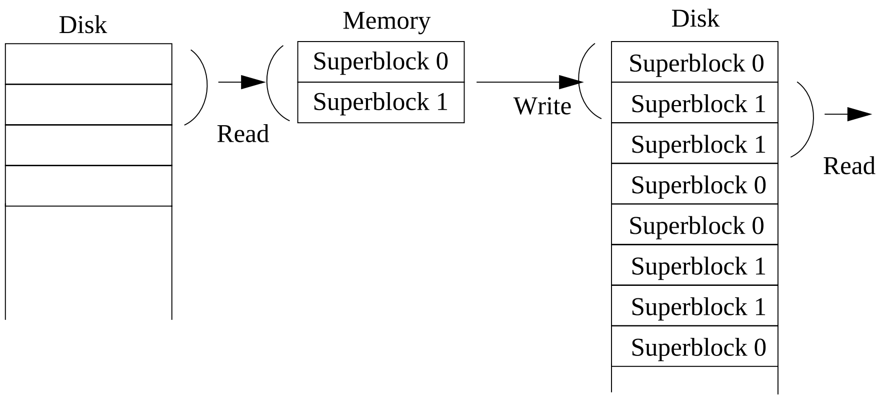
3.3.1 Overview

One of the key features of our algorithm is the reduction in the total number of I/O operations, which is achieved by means of an efficient data layout scheme on the disk. For example, in [15], there are three I/O operations (one read operation and two write operations) in each step when $M = 2N$ and $B = N$ (see Figure 17). The concept of a step is explained in detail in Section 3.3.2. Our algorithm requires only a single write operation in each step as against two write operations in the case of the previous algorithm in [1, 15]. Since our algorithm consists of only the same number of steps as the previous algorithms, there is a considerable reduction in the total number of write operations.

This reduction in write operations is a consequence of the efficient data layout scheme ($L_s, 0 \leq s \leq t-1$) employed. The proposed layout scheme provides a means for reducing the number of write operations while maintaining the same number of read operations. Thus, the number of I/O operations is reduced from three to two in each step which leads to a 33% reduction in the total number of I/O operations.



(a) Previous Approach [12]



(b) Our Approach

Figure 17: An illustrative example (r_s = 2)

Another technique used in our algorithm is balancing the numbers of read and write operations. In balancing the numbers of read and write operations, the key idea is that the total number of I/O operations can be reduced by reducing the number of write operations at the expense of an increased number of read operations. For example, when $r_s = 32$, in each step, the number of read (write) operations in [15] is 1 (32), where r_s is explained in Section 3.3.2. In our algorithm, we increase the number of read operations to 9 in order to reduce the number of write operations to 9. This results in a 45% reduction in the total number of I/O operations. Note that a straightforward method to balance the number of read and write operations reduces the total number of I/O operations by only one (see

Section 3.3.2). The data that is written onto the disk in z write operations in the previous algorithm is written in one write operation in our algorithm (see Figure 18). Thus, there is a reduction in the number of write operations by a factor of z. However, this writing causes the data to be "scattered": consecutive superblocks are not contiguous as shown in Figure 18. The superblock is a chunk of memory of size M/r_s at $s^{theorem}$ stage, $0 \leq s \leq t-1$, and is explained in detail in Section 3.3.2. In a subsequent read operation, to read the data that is scattered, z read operations are needed. By choosing an optimal value of z, the total number of I/O operations is reduced.

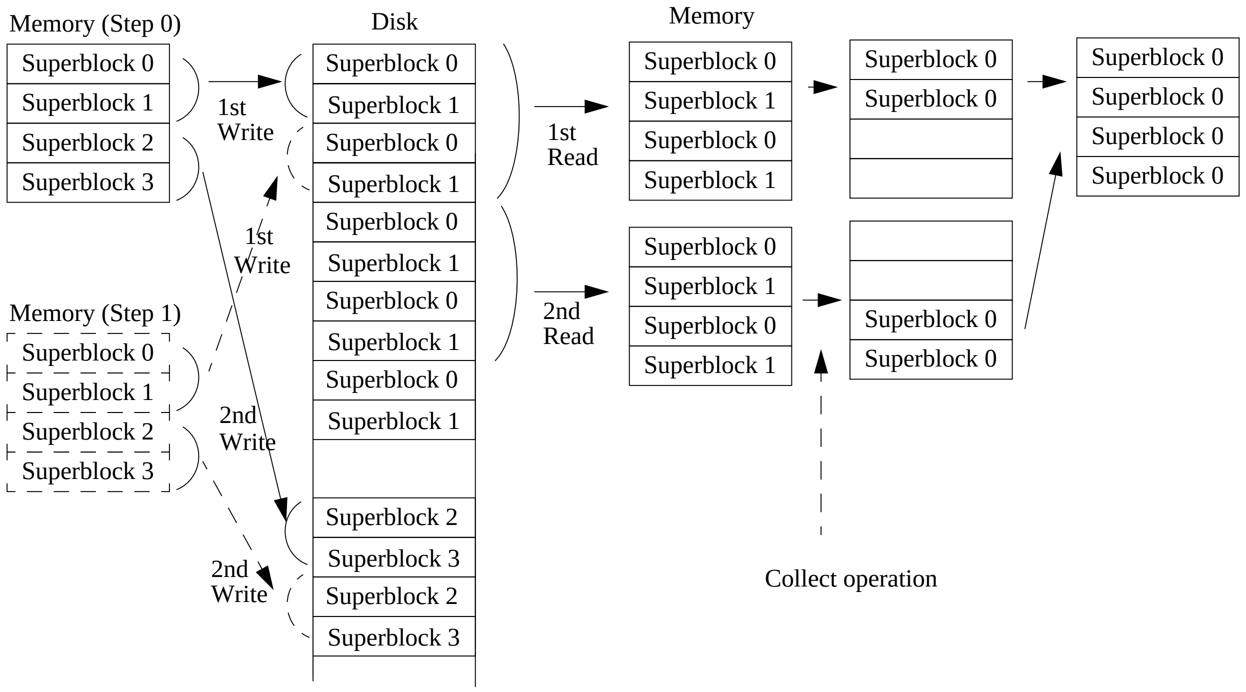


Figure 18: An illustrative example ($r_s = 4$ and $z = 2$)

The index computation takes up a significant portion of the total execution time. In the previous algorithms [1, 15], the entire available memory is used for reading data from disk. Even though this approach maximizes the memory utilization, it results in excessive index computation cost. (Index computation refers to computing the source or destination addresses of each data.)

To eliminate the index computation cost, the available memory is partitioned into two different-size buffers (read and write buffers). Instead of performing a permutation before every write operation, only the data needed for each write operation is moved into the write buffer. This is denoted as a $collect$ operation. The stride of the data access for the collect operation is constant. Thus, it can be performed using inexpensive do-loops.

If the same schedule as in the previous algorithms is used (collect operations followed by write operations), then the size of the write buffer should be $M/2$. However, in our algorithm, the utilization of the write buffer is increased using our schedule which results in a smaller write buffer. In

```
1  **for** $s$ = 0 to $t$-1 // for each stage
2      **for** $step$ = 0 to $N^2/M$-1 // for each step
3          Read data from disk using the layout $L_{s-1}$;
4          Permute the data on memory;
5          Write data to disk using the layout $L_s$;
```

Figure 19: Reducing the number of I/O operations

our schedule, a write operation follows each collect operation. Since the read buffer size is less than the available memory size, the number of I/O operations is increased slightly. However, as shown in Section 5, the total execution time is reduced significantly due to reduction in the index computation time.

3.3.2 Details

Additional details of our algorithm as well as the analysis are presented in this section. However, due to space limitation, proofs of the theorems are not included.

3.3.2.1 Reducing the Number of I/O Operations Our algorithm to reduce the number of I/O operations is elaborated here (see Figure 19). Note that the matrix size is $N \times N$ and $N = \prod_{s=0}^{t-1} r_s$.

The algorithm consists of t stages (Line 1). In the $s^{theorem}$ stage, $0 \leq s \leq t-1$, a $submatrix$ is defined as follows. Let us denote the data at row i and column j in the original input matrix as $d_{i,j}$. A submatrix, $S_{k,l}$, $0 \leq k, l \leq R_s - 1$, consists of $d_{i,j}, kN/R_s \leq s \leq (k+1)N/R_s - 1, lN/R_s \leq j \leq (l+1)N/R_s - 1$, where $R_s = \prod_{s=0}^{s} r_s$.

In each stage, there are N^2/M steps (Line 2). In each step, the data is first read into memory (Line 3). The data in the memory that is in the same submatrix is moved to a contiguous chunk of the memory (Line 4). Let us denote this contiguous chunk of memory as a $superblock$. There are r_s superblocks of size M/r_s. The superblocks are written onto the disk (Line 5). The layout, $L_s, 0 \leq s \leq t-1$, specifies the locations of the superblocks on the disk.

The layout, the schedule of reading data from the disk, and the schedule of writing data onto the disk are explained in the following four cases. Case 1 and Case 2 pertain to the cases where as much data as the memory size can be read from the disk or written onto the disk in one I/O operation ($B = M$). Our analysis shows that efficient data arrangement reduces the number of I/O operations by a factor of $(r_s + 1)/r_s$ (Case 1). In addition to this, if $r_s \geq 8$ (Case 2), balancing the number of I/O operations further reduces the total number of I/O operations.

If $M/r_s < B < M$ (Case 3), our algorithm provides the best performance compared with the previous algorithms. In the other case, $B \leq M/r_s$ (Case 4), our algorithm has the same performance as the previous algorithm in [1] with respect to the number of I/O operations.

Note that reducing the index computation time (discussed in Section 3.3.2.6) further improves the performance in all the cases.

3.3.2.2 Case 1: ($B = M$ and $1 < r_s < 8$) The key idea here is data arrangement on the disk, L_s. The matrix is first partitioned into R_{s-1} areas. Each area includes N/R_{s-1} rows. The layout and schedules of reading and writing data in each area are explained for two cases.

If $r_s = 2$, the layout, L_s is as follows: the first superblock is stored in the $j^{theorem}$, $\lfloor(j + 1)/2\rfloor \bmod 2 = 0$, superblock and the second superblock is stored in the rest of the superblocks (see Figure 20). The number in each small square denotes a data element. Notice that the data is in row-major order in the initial matrix at stage 0 and in column-major order in the last matrix in stage 2. Using this layout, in the first step, the first and second superblocks are stored on the disk in one write operation since they are contiguous in the memory as well as on the disk. This is illustrated in Figure 20. In the figure, at each stage, the left (right) matrix is initial (final) matrix. In the second step, the third and fourth superblocks are saved on disk, and so on. In the next stage, the data in the second and third superblocks are read into the memory using one read operation, and data in the fourth and fifth superblocks are read into the memory in the next step.

If $r_s > 2$, in the $st^{theorem}$ step, two superblocks, $(st + 1) \bmod r_s$ and $(st + 2) \bmod r_s$, are stored in one write operation and the rest of the data is written in $(r_s - 2)$ write operations which results in $r_s - 1$ write operations (see Figure 21). The figures in the middle show the data in memory after permutation.

A comparison of the numbers of I/O operations in the algorithm in [15] and our algorithm is shown in Table 5.

Table 5: Number of I/O operations in each step

r_s	2	3	4
Kaushik's Algorithm [15]	3	4	5
This Paper	2	3	4
Reduction	33%	25%	20 %

3.3.2.3 Case 2: ($B = M$ and $r_s \geq 8$) In this case, the total number of I/O operations can be further reduced by balancing the numbers of read and write operations in addition to the data layout and the schedule explained in Case 1. In Kaushik et al.'s algorithm, the difference between the numbers of read and write operations is large. That is, in each step, the number of read operations is 1 and the number of write operations is r_s. In our algorithm, we develop a technique that reduces the number of write operations at the expense of an increased number of read operations.

Note that if a straightforward method is used, the number of write operations is reduced to $r_s - z$, where z is the number of the new read operations. Then, the new total number of I/O operations is $(r_s - z) + z = r_s$. The total number of I/O operations is reduced by only one. In our algorithm, we decrease the number of write operations to approximately r_s/z. Then, the total number of I/O operations can be reduced by choosing an optimal value of z.

In the previous algorithms, each superblock is stored in one disk write operation. In our algorithm, z blocks are stored on the disk in one write operation. Thus, the number of write operations is reduced by a factor of z. In each read operation, to read data that is "scattered" in noncontiguous locations, we need to perform z read operations. It can be shown that the optimal value of z is $\sqrt{2r_s}$ in the $s^{theorem}$ stage, $0 \leq s \leq t-1$.

The total number of I/O operations in the algorithm in [15] and in our algorithm are compared in Table 6. The algorithm in [1] is not compared here since it cannot be used in this case. The following Theorem 1 applies to Case 1 and Case 2.

Theorem 1 *In the Linear Model, the total number of I/O operations in our algorithm is* $\frac{N^2}{M}\sum_{s=0}^{t-1}\min(r_s, \sqrt{2r_s}+1)$.

3.3.2.4 Case 3: ($M/r_s \leq B < M$) This is similar to Case 2; the only difference is the size of the block. It relaxes the restriction ($r_s \leq M/B$) that was imposed in [1]. In our algorithm, we can increase the value of r_s to be larger than M/B so that the number of stages is decreased. The optimal value of z is Br_s/M.

Theorem 2 *In the Parallel Disk Model, the total number of I/O operations in our algorithm is* $\frac{2N^2}{M}\sum_{s=0}^{t-1}(\sqrt{\frac{r_s M}{B}}+1)$*, where* $\frac{M}{r_s} < B < M, 0 \leq s \leq t-1$.

3.3.2.5 Case 4: ($B \leq M/r_s$) In this case, our algorithm is the same as in [1].

3.3.2.6 Reducing Index Computation Time In the previous algorithms, the available memory is fully utilized to reduce the number of I/O operations. In other words, in a read operation, as much

Table 6: Number of I/O operations in each step

r_s	r_s = 32		r_s = 128	
	Kaushik's Algorithm	Our Algorithm	Kaushik's Algorithm	Our Algorithm
# of Read operations	1	9	1	17
# of Write operations	32	9	128	17
Total	33	18	129	34

data as the size of the memory is read from disk. However, this results in a large index computation time. Permuting the data within the memory requires destination location of each data element to be computed.

To reduce the total execution time, we eliminate the expensive index computation by using the algorithm shown in Figure 22. In our algorithm, we partition the memory into two different-size buffers: one with size M_r (Read buffer) and the other with size M_w (Write buffer). The read buffer is used for reading data from disk. After reading the data, there are r_s/z sets of collect and write operations, where z is a positive integer and explained in Section 3.3.2.6. In each collect operation, data in z supermatrices is collected in the write buffer. The sizes of the write and read buffers are determined as $Mz/(r_s + z)$ and $MR_s/(r_s + z)$, respectively.

In the collect operation, the data in the z superblocks is located in $M_r R_{s-1}/N$ chunks of data. The amount of the data in each chunk is N/R_{s-1}. Thus, to collect the data in z superblocks, multiple-level do-loops are necessary. In each do-loop, the required computations are simple additions to compute loop-variables. Note that, in the previous algorithms [1, 15], the required computations for permutation of the data consist of both the index computations and the loop-variable computations. In our algorithm, since the loop-variables are used to collect data to the write buffer, the index computation is eliminated.

The collected data in the write buffer is written onto the disk in a write operation (Line 6). Even though the number of I/O operations increases by a factor of M/M_r, the total execution time is reduced significantly due to the elimination of index computation time.

3.4 Experimental Results

We implemented the algorithms on a DEC Alpha system (Cray T3E) at the San Diego Supercomputing Center (SDSC) and a Sun Enterprise 4000 system at the University of Southern California. For comparison purposes, Kaushik et al.'s algorithm described in Section 3.2.3 was also implemented.

Note that Aggarwal et al.'s algorithm described in 3.2.2 has the same total execution time as the Kaushik et al.'s algorithm in our experiments. Even though the two algorithms perform permutation using different methods and the data being permuted are different, the permutation times are the same. If the block size is smaller than M/r_s at the $s^{theorem}$ stage, $0 \leq s \leq t-1$, then both the algorithms require the same I/O time, where t is the number of stages. I/O time is different for the two algorithms when the amount of data transferred in one I/O operation is smaller than B. The amount of the data transferred in one I/O operation in our experiments ranges from 128 KBytes to 2 MBytes and the typical size of B in state-of-the-art platforms is 4 KBytes. Thus, the performance of the two algorithms are the same in our experiments. Therefore, the execution time reported under the heading "previous algorithm" refers to both the algorithms.

The system parameters of the computing platforms in which our experiments were conducted are summarized in Table 7.

The amount of main memory allocated to the data was varied from 16 MBytes to 64 MBytes and

Table 7: Computing Platforms on which experiments were conducted

Platform	Cray T3E	Sun Enterprise
Processor	DEC Alpha (300 MHz)	UltraSPARC (336 MHz)
OS	UNICOS/mk 2.0.3.39	SunOS Release 5.6 Version Generic_105181-11
Compiler	Cray Standard C Version 6.1.0.1	SC4.0

the data size was varied from 512 MBytes to 2 GBytes. For each parameter value, the algorithms were executed 5 times and the maximum, average, and minimum values were calculated. The speedup of our algorithm over the previous algorithms was calculated for each parameter setting. The results of our experiments are shown in the figures from Figure 23 to Figure 28. The results show that our algorithm reduces the execution time by about 50%.

The execution times correlate well with our analysis. From our experiment in implementing the previous matrix transpose algorithm on the Sun Enterprise, the time for I/O is 10 nsec/byte, the time for index computation is 25 nsec/byte, and the time for data movement is 20 nsec/byte. In this case, the time for I/O is measured as the elapsed wallclock time, which is the most tangible method of measuring I/O performance. The time for I/O is obtained by dividing the total elapsed time by the data size. The time for index computation involves the computation of either the source or the destination address. The destination address y of a data located in x is based on the equation is $y = \frac{x}{a} + xmodb$, where a and b are variables calculated based on the step and stage. Thus, a single index computation involves two divisions and a multiplication, rendering it an expensive operation. For example, on the DEC Alpha 21264, an integer multiplication takes 13 cycles and integer divide is not directly supported. Thus, when the data size is 2 GBytes, the expected total execution time is 2 G $\times$ 3 stages $\times$ 2 (read and write) $\times$ (10+25+20) nsec = 660 sec which is similar to the actual 642 sec. For the size of 512 (128) MBytes, the expected time is 165 (41) sec and actual time is 148 (37) sec.

The execution time is increased about four times as the data size is increased four times. This is reasonable since the three major costs (I/O time, index computation time, and memory-memory transfer time) are proportional to the data size. Thus, we expect the same speedup for the larger data sizes than 2 GBytes.

3.5 Further Extensions

Our matrix transpose algorithm can be extended to the more general problem of Bit Matrix Multiply Complement (BMMC) [7]. In this problem, we consider a matrix X of size $N \times N$. The permutation of the data is represented using a nonsingular matrix A of size $2\log N \times 2\log N$ and a vector c, where each of the entries of A and c is a binary number. The address of an element is represented in bit notation. A target address is $y = (y_0, y_1, ..., y_{2\log N-1})$, and source address is $x = (x_0, x_1, ..., x_{2\log N-1})$, where the first $\lg N$ bits represent the row number and next $\lg N$ bits represent the column number. The target address is obtained from the source address by $y_i = (\bigoplus_{j=0}^{2\log N-1} a_{i.j}x_j) \oplus c_i, 0 \leq i \leq 2\log N - 1$, where the $\oplus$ denotes an exclusive-or operation. A specific case of BMMC is a matrix transpose.

The BMMC consists of many steps. In each step, there are three basic operations as in the case of matrix transpose: read data from disk, permutation of the data on memory, and write data onto disk. Since the key ideas in our matrix transpose algorithm (reducing the number of I/O operations and index computation) are independent of the permutation method of the data in memory, our algorithm will greatly enhance the computational efficiency of the BMMC problem.

To reduce the number of I/O operations, the algorithm in Figure 19 is used to employ the two techniques: efficient data layout scheme and balancing the number of I/O operations. The permutation of the data in memory is performed as in [7]. To reduce the index computation time, the algorithm in Figure 22 is used: the available memory is partitioned into two buffers, the permutation is replaced by collect operations, and the collect operations and write operations are scheduled to maximize the memory utilization.

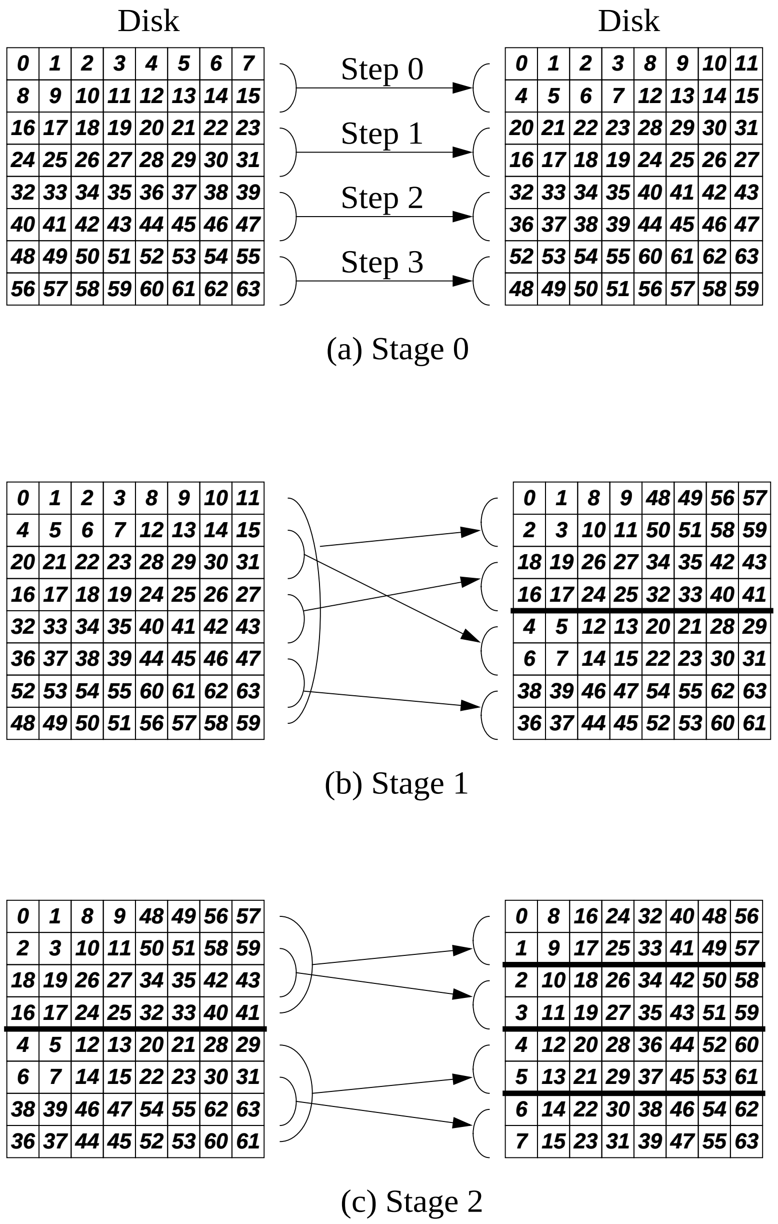
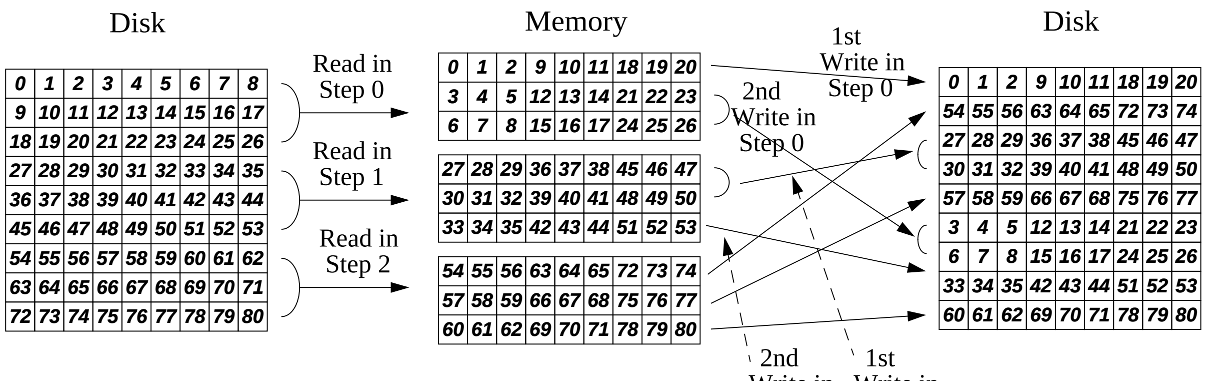
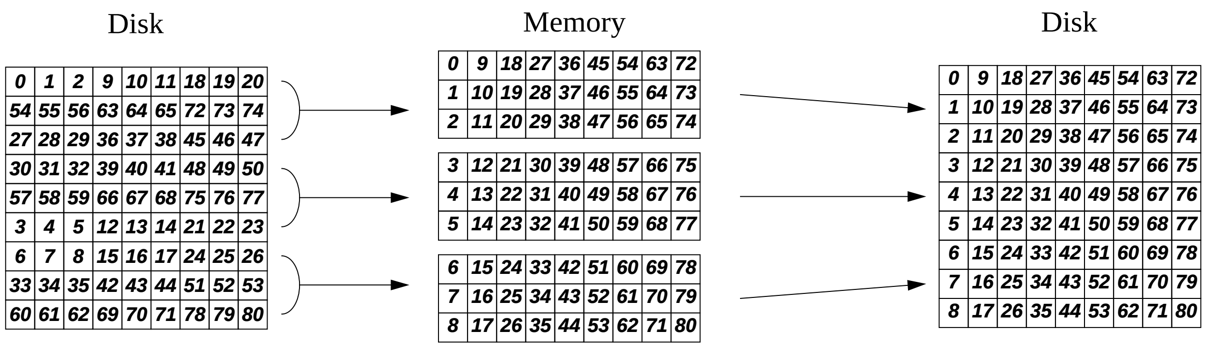


Figure 20: An illustrative example ($N = 8 = \prod_{s=0}^{2} 2$, and $M = 16$)



(a) Stage 0

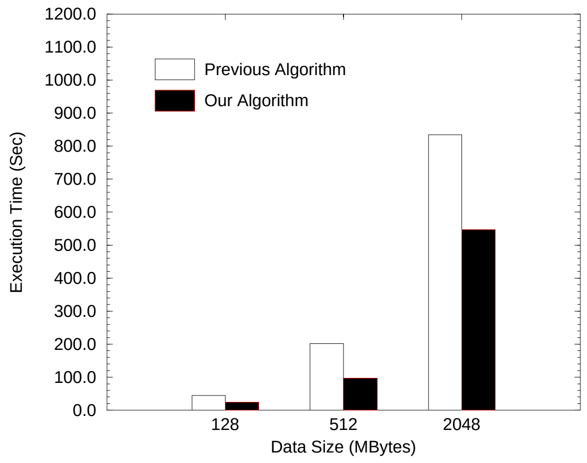


(b) Stage 1

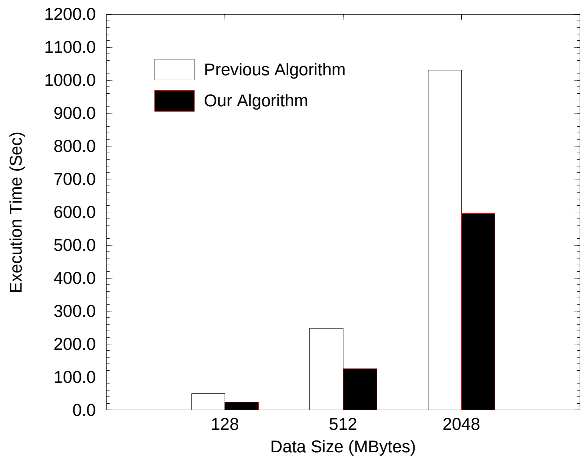
Figure 21: An illustrative example (N = 9 = $\prod_{s=0}^{1}$ 3, and M = 27)

```
1   for $s$ = 0 to $t$ - 1 // for each stage
2       for $step$ = 0 to $N^2/M_r$ - 1 // for each step
3           Read data from disk;
4           for $i$ = 0 to $r_s/z$ - 1
5               Move data that has the $i^{theorem}$ supermatrix to the write buffer;
6               Write data in write buffer to disk;
```

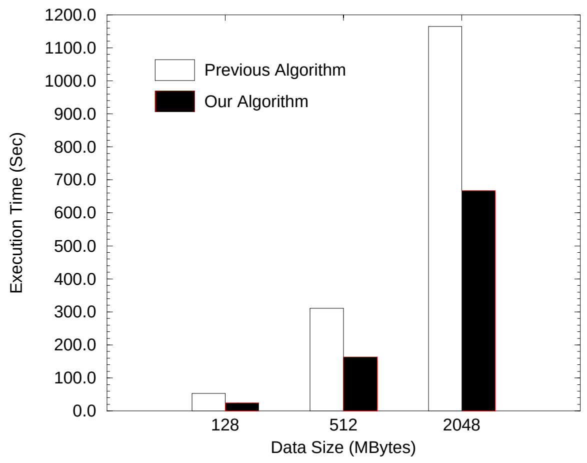
Figure 22: Pseudo-code for our algorithm



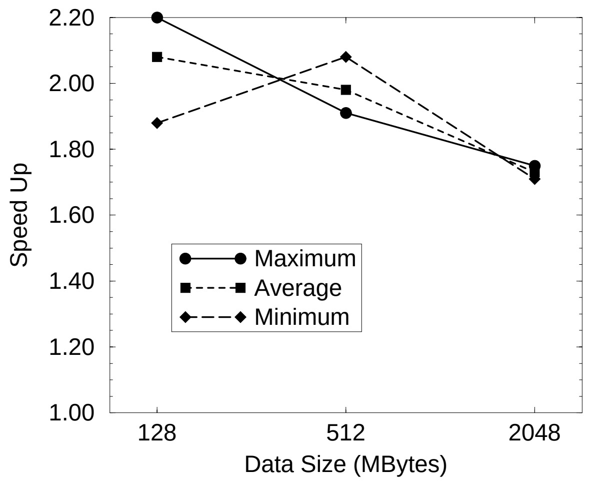
(a)



(b)

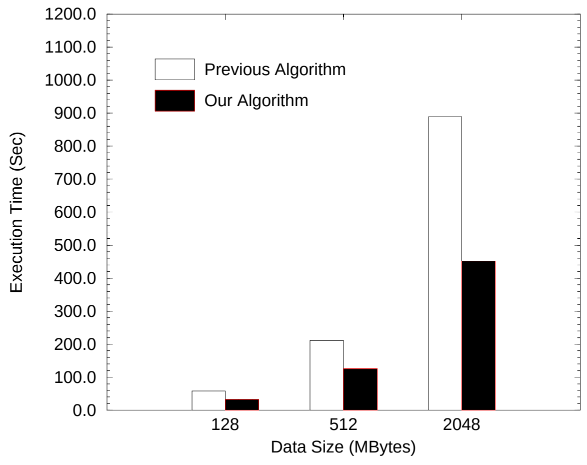


(c)

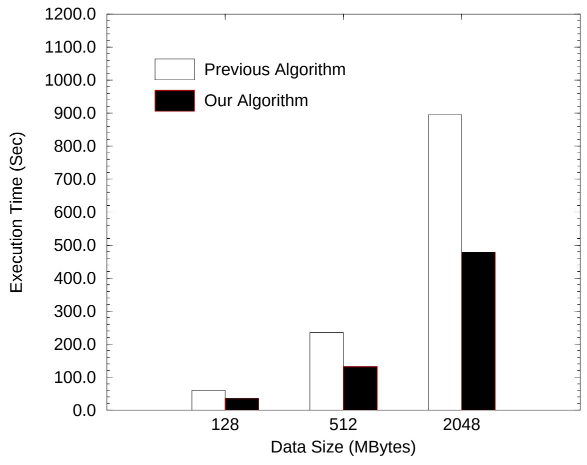


(d)

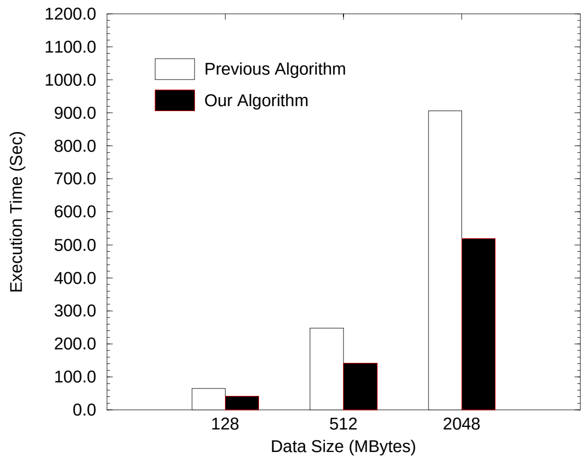
Figure 23: Experimental Results on DEC Alpha (T3E) (a) Minimum (b) Average (c) Maximum (d) Speedup, M = 16 MBytes



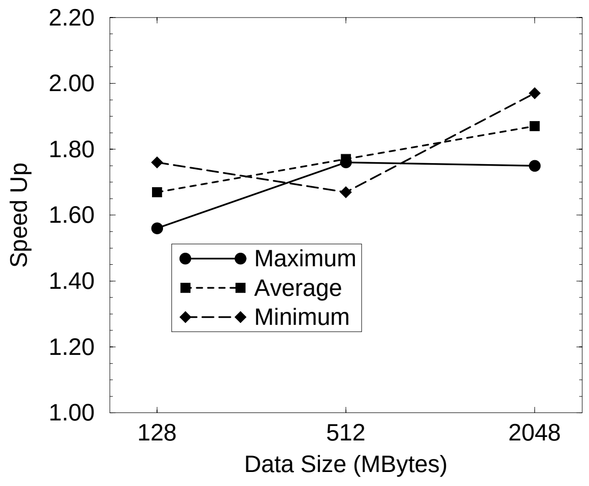
(a)



(b)

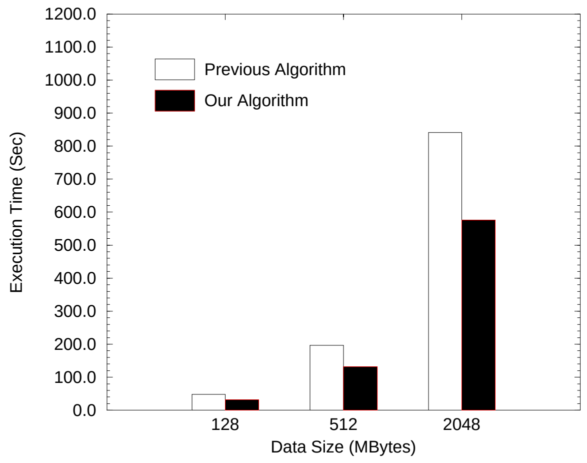


(c)

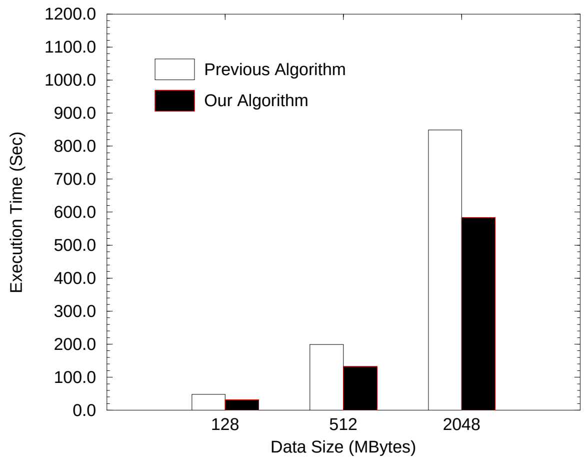


(d)

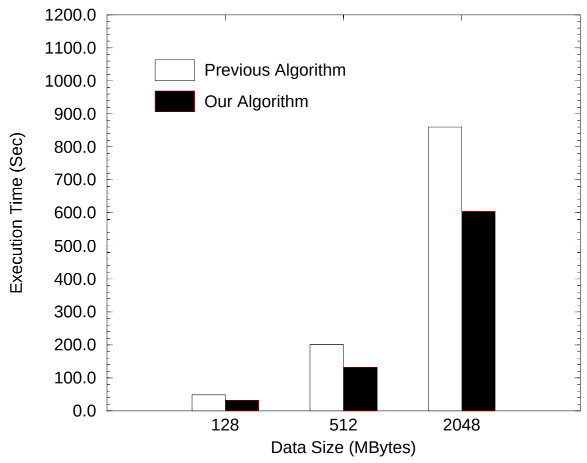
Figure 24: Experimental Results on DEC Alpha (T3E) (a) Minimum (b) Average (c) Maximum (d) Speedup, M = 32 MBytes



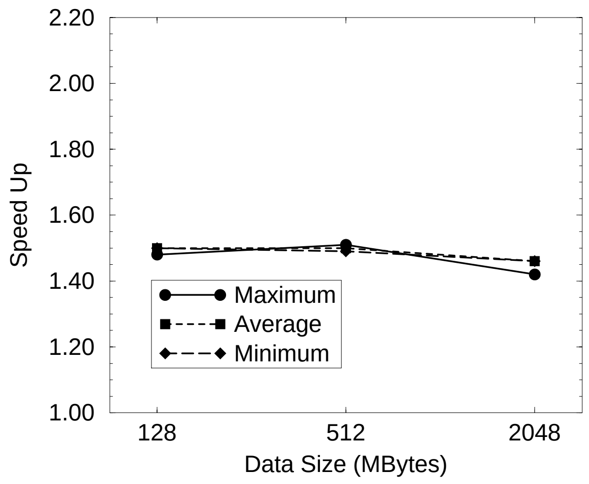
(a)



(b)

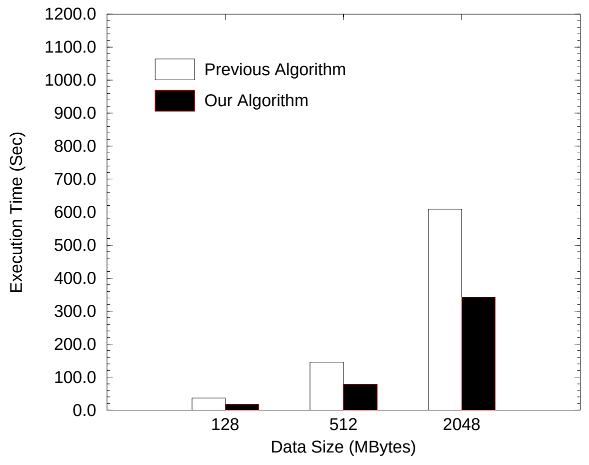


(c)

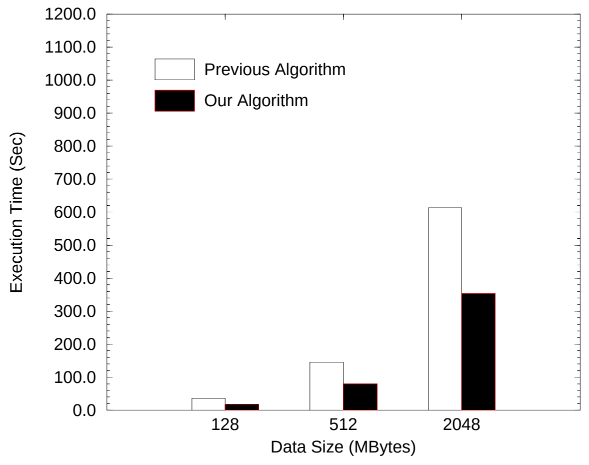


(d)

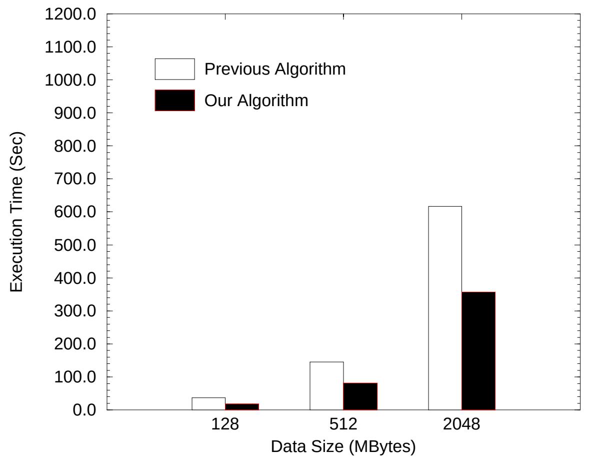
Figure 25: Experimental Results on DEC Alpha (T3E) (a) Minimum (b) Average (d) Speedup, M = 64 MBytes



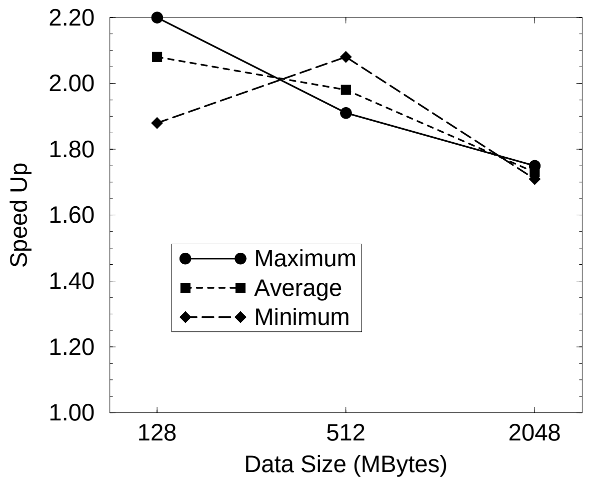
(a)



(b)

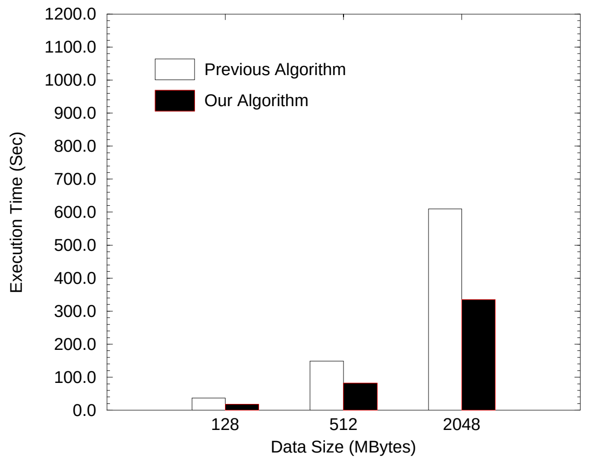


(c)

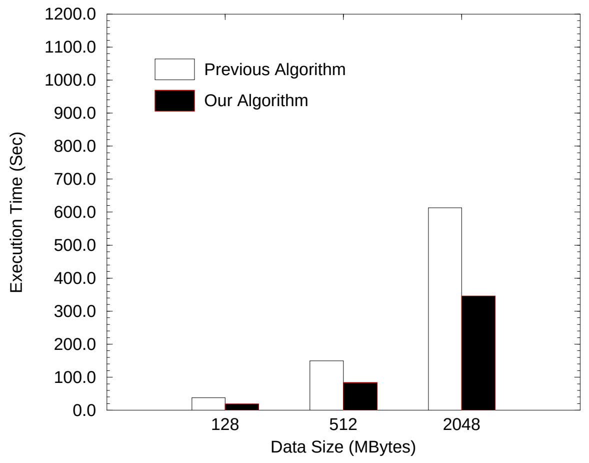


(d)

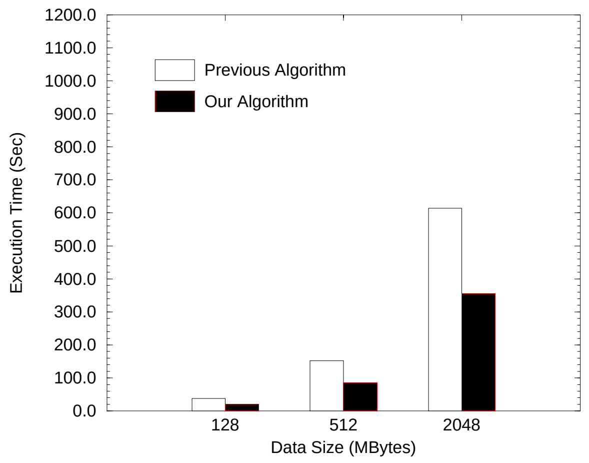
Figure 26: Experimental Results on Sun Enterprise (a) Minimum (b) Average (c) Maximum (d) Speedup, M = 16 MBytes



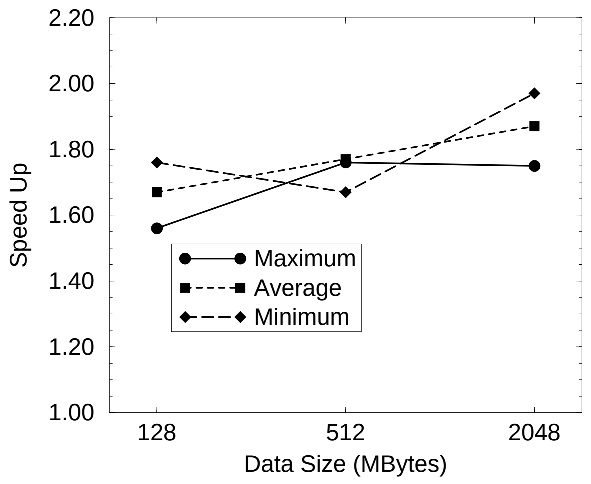
(a)



(b)

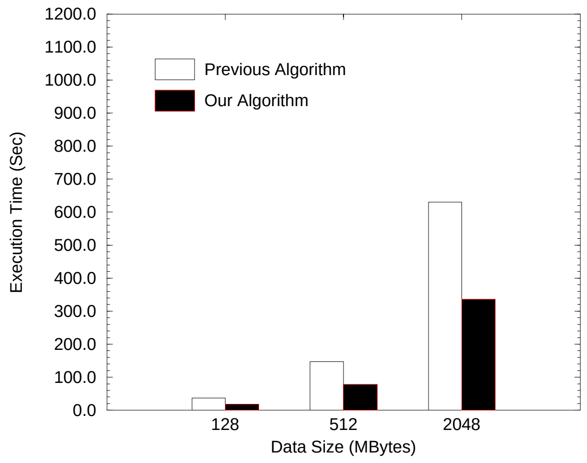


(c)

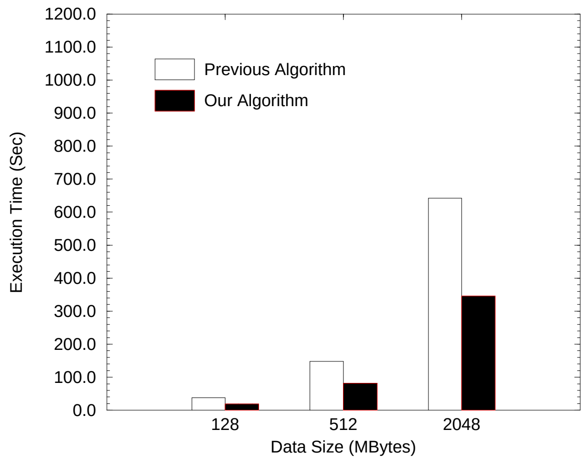


(d)

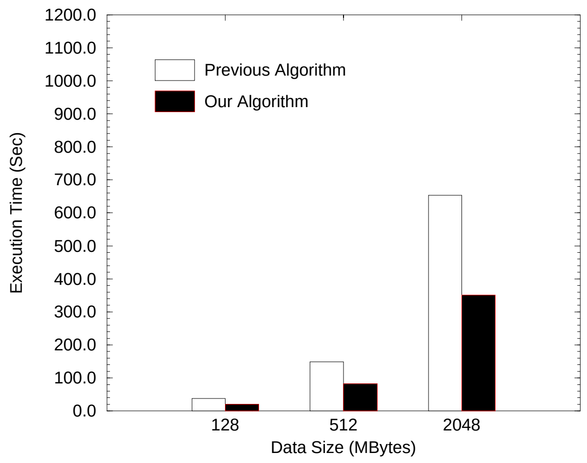
Figure 27: Experimental Results on Sun Enterprise (a) Minimum (b) Average (c) Maximum (d) Speedup, M = 32 MBytes



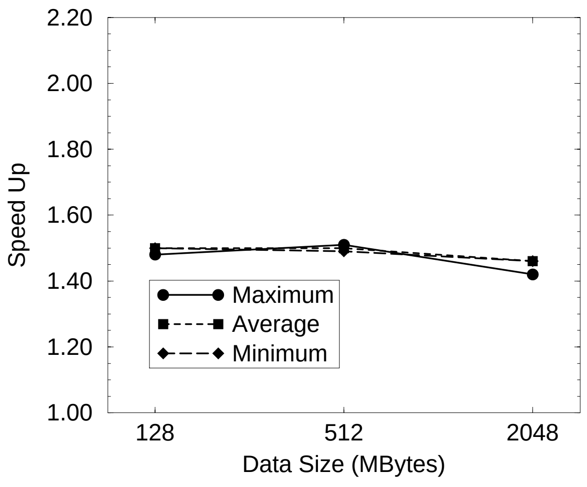
(a)



(b)



(c)



(d)

Figure 28: Experimental Results on Sun Enterprise (a) Minimum (b) Average (c) Maximum (d) Speedup, M = 64 MBytes

4 Matrix Multiplicaton

In this section, we describe cache conscious data layouts for efficient matrix multiplication. We propose a novel data layout for efficient matrix multiplication. For standard matrix multiplication, we show how to reduce cache pollution, data cache misses and TLB (Translation Look-Aside Buffer) misses. We start with a description of data access costs in the memory hierarchy and issues in data layout design. In Section 4.3 we present an efficient data layout for the standard matrix multiplication operation. As seen from the experimental results obtained from three machines, our approach compares favorably with the three others that we considered.

4.1 Data Access in the Memory Hierarchy

In general computer systems, each processor node communicates with its cache memory (fast memory), which in turn communicates with the local main memory. The processor may also access the main memories of other processors (remote memories). If the data requested by the processor is not available in cache, a cache miss occurs. During a cache miss, data is fetched from the main memory into the cache. Typically, the main memory access is about 10 times slower than cache memory. For instance, on a DEC Alpha 21154 platform, the on-chip cache access latency is around 10 nsec, while the main memory access latency is 253 nsec. For better performance, the frequency of cache misses should be kept low.

To reduce memory latency, the processor attempts to fetch data from the cache memories. Physical memory is organized into equal sized pages (say 4k bytes). All addresses within a physical memory page lie in consecutive memory locations. The virtual memory is also organized into pages of the same size. Each virtual page is assigned a page number. When main memory is out of space, one page is brought from the disk at a time. Before the processor can execute a memory access instruction, the virtual address has to be translated. Translation determines the physical page number corresponding to a given virtual page number. This mapping information is provided by the page tables which are the data structures stored in the main memory.

It is very expensive to look up the page tables for each address. To reduce the cost of translation, a special high-speed cache is provided to buffer the page table entries. This cache is called the Translation Lookaside Buffer (TLB). In Figure 29, the CPU looks up the TLB for each address translation. Usually, the TLB size is limited to only a few entries. In case the TLB misses, the missing translation entry is loaded into it from the page table. A TLB miss significantly increases the translation time.

4.2 Issues in Design of Data Layout

A data layout is the scheme in which data elements are assigned addresses in the memory. In a row major layout (See Figure 30), elements in one row are assigned consecutive memory locations. In a column major layout elements in one column are assigned consecutive memory locations.

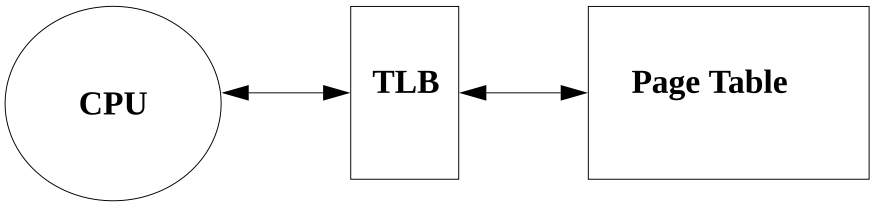


Figure 29: Using TLB for fast page number translation

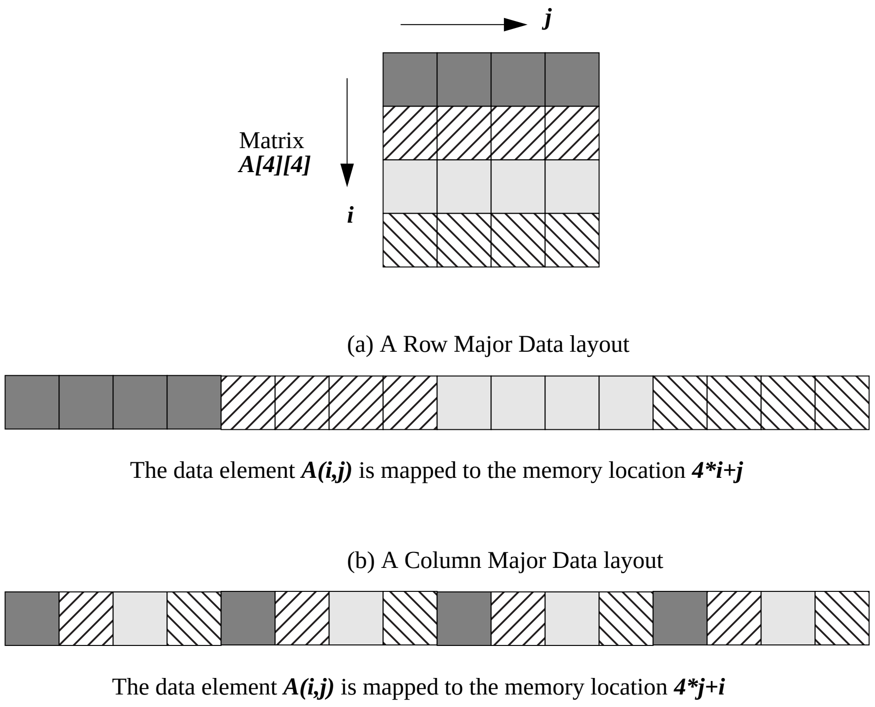

j
Matrix
A[4][4]
i
(a) A Row Major Data layout
The data element A(i,j) is mapped to the memory location 4*i+j
(b) A Column Major Data layout
The data element A(i,j) is mapped to the memory location 4*j+i

Figure 30: Example data layouts

Mismatch between data access patterns and data layout patterns can increase the occcurance of cache misses. Consider the example shown in Figure 31, where a cache with one 4 word sized block has data laid out in row major order. A row major access pattern causes only 4 cache misses since each cache miss loads the entire row into a block. In case of column major access, the number of cache misses goes up to 16. This happens because each cache miss loads one useful element and three unused elements.

4.3 Data Layout for Standard Matrix Multiplcation

Large scale matrix multiplication deals with matrices which do not fit into the cache. Conventional methods of matrix multiplication are not cache-friendly. In the matrix multiplication $C = A \times B$, elements in Matrix B are accessed in column major order. Assuming that both A and B are stored in

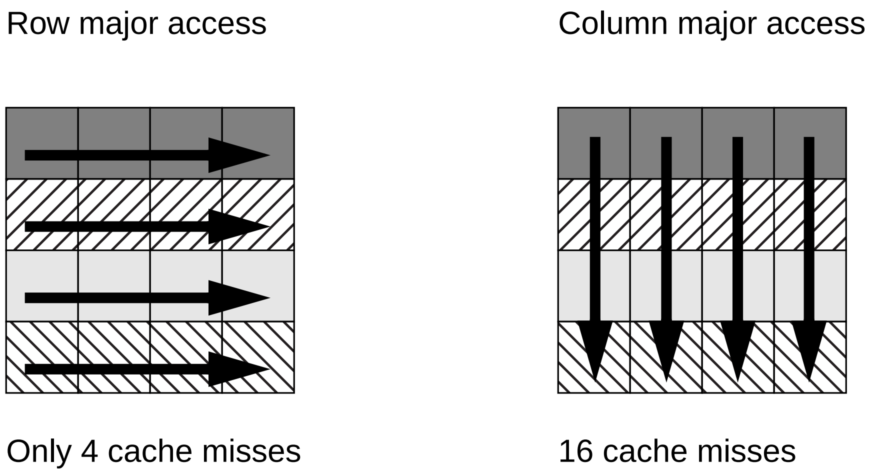


Figure 31: Access of page numbers using translation look-aside buffer

row major order, each access to B results in a cache miss since the consecutively accessed elements are located far apart in memory. Elements of B are repeatedly accessed when computing different elements of C, but they do not remain in the cache for reuse as the cache capacity is small. Besides, only small portions of the fetched cache blocks are accessed before they get replaced due to conflicts. The net result is a large number of cache misses.

Blocking is widely used to reorder the computation sequence, and thereby, reduce cache pollution. Blocking exploits temporal locality to reduce cache misses. However, cache conflict misses still exist in blocking based computation. TLB thrashing is another possible problem for large matrices. Elements of the same block can span several different pages when a column is accessed. This may cause a spate of misses in the small-sized TLB. Therefore, blocking does not offer a complete solution for cache-optimized matrix multiplication.

One of the key ideas of our approach is to reorganize the layout of matrix data stored in the main memory such that it is cache friendly. We perform this reorganization prior to computation. In the proposed data layout, we transpose matrix B such that the data layout matches the data access pattern. This reduces cache pollution to a considerable extent. We partition each matrix into square sub-matrices, denoted as blocks. Elements belonging to the same block are stored in consecutive memory locations in row major order. The resulting data layout is shown in Figure 32 (b). There are no conflict misses among the elements in the same block. The block size is chosen to be equal to the virtual page size so that computations within a block will not result in a TLB miss.

We have shown using the SimpleScalar simulator (a state-of- the-art architecture simulator) that the total number of TLB misses is reduced significantly with our layout compared with the standard row major layout. The above data layout transformation (i.e., matrix transpose and block data layout) is performed once. Thus, no additional overhead is incurred during the computation.

Our proposed layout reduces cache pollution, cache misses, and TLB misses without excessive overheads like reorganizing the data in memory or computing indices.

We have implemented our scheme on UltraSPARC II, Alpha 21264, and Pentium III based machines for matrix sizes ranging from 1024x1024 to 1536x1536. Tables 8, 9, and 10 compare the performance of our scheme with the naive CBLAS (without blocking), CBLAS (with blocking), and CBLAS (with blocking and copying) algorithms on three machines. The reported execution times are wall clock times. On UltraSPARC II, Alpha 21264, and Pentium III machines, we used the gcc compiler wit ”-O3” optimization option. As the experimental results show, our scheme is up to 15 times faster than naive CBLAS, 2 times faster than blocking based CBLAS, and is superior to blocking and copying based CBLAS implementations on UltraSPARC II. On Alpha 21264, our scheme performs up to 5 times faster than the naive CBLAS, up to 3 times faster than blocking based CBLAS, and is faster than blocking and copying based CBLAS implementation. On Pentium III, our scheme outperforms all the three previous techniques.

Table 8: Execution time on UltraSPARC II (400 MHz, 2MByte L2 cache)

Matrix size	CBLAS (Native)	CBLAS (Blocking)	CBLAS (Blocking +copying)	Our Algorithm
1024×1024	243.418	34.147	22.271	17.240
1200×1200	370.387	40.663	34.478	29.920
1280×1280	455.795	65.952	42.262	33.842
1400×1400	592.934	66.675	53.522	45.192
1536×1536	810.280	124.489	74.740	60.865

Table 9: Execution time on DEC Alpha 21264 (500MHz, 4MByte L2 cache)

Matrix size	CBLAS (Native)	CBLAS (Blocking)	CBLAS (Blocking +copying)	Our Algorithm
1024×1024	125.237	23.214	16.427	13.283243.418
1200×1200	194.556	28.330	28.465	22.383370.387
1280×1280	238.613	31.115	29.984	26.503455.795
1400×1400	310.947	44.730	45.311	35.248
1536×1536	415.870	79.907	54.045	45.816

Table 10: Execution time on Pentium III (450MHz, 512KByte L2 cache)

Matrix size	CBLAS (Native)	CBLAS (Blocking)	CBLAS (Blocking +copying)	Our Algorithm
1024×1024	92.566	27.136	22.030	17.335
1200×1200	152.311	30.107	33.390	28.050
1280×1280	184.973	52.325	43.117	34.184
1400×1400	244.652	48.215	55.644	45.756
1536×1536	325.241	90.306	74.345	59.137

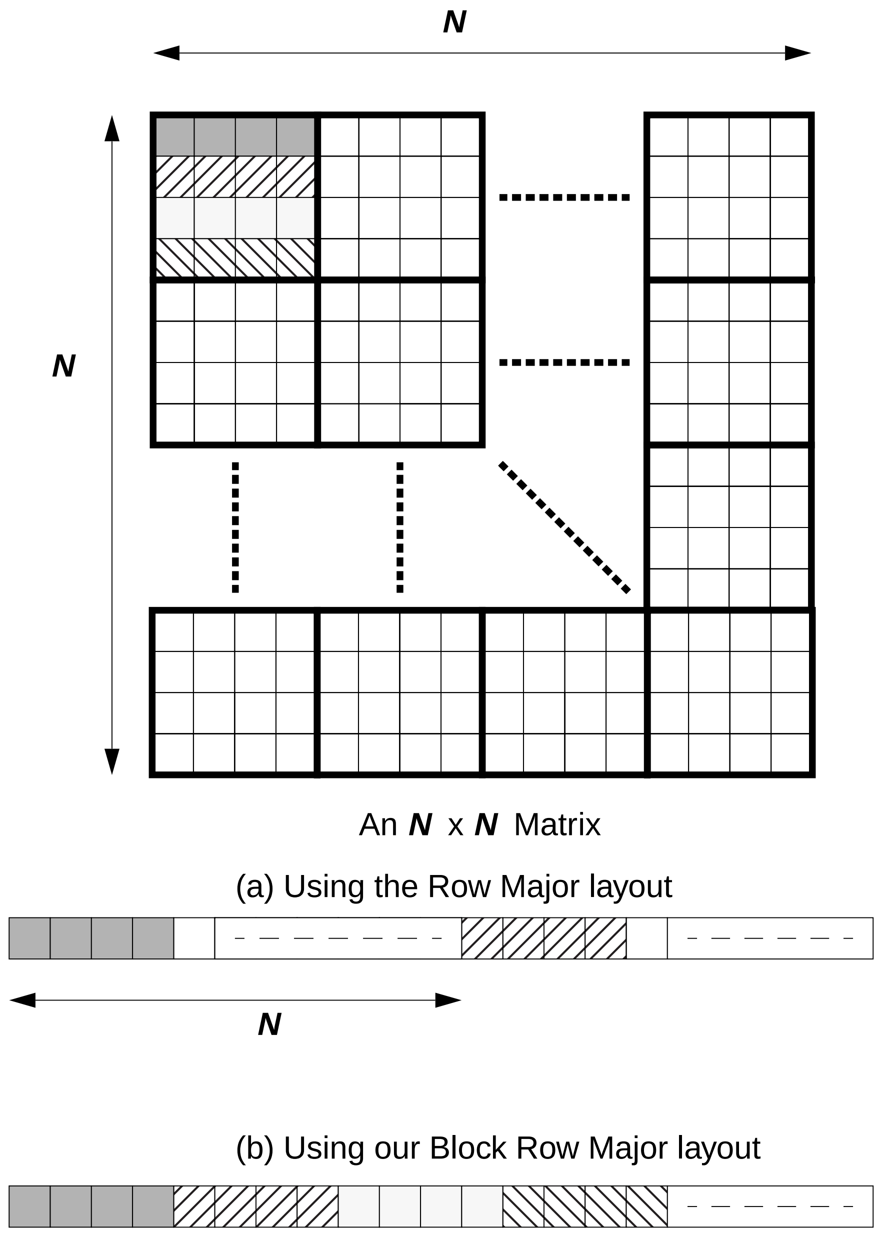

N
N
An N x N Matrix
(a) Using the Row Major layout
N
(b) Using our Block Row Major layout

Figure 32: Illustration of our proposed block data layout for matrix multiplication

5 Finite Element Method

In this section, we present an efficient data layout for mesh generation of generic two-dimensional Finite Element Methods (FEMs). FEMs are widely used for numerical analysis and simulation of a large variety of scientific problems. The first step of a typical FEM is to generate an appropriate mesh of the problem domain. FEMs for grand challenge applications can have 10,000 to 1,000,000 elements. Often very large numbers of memory accesses are required. We describe a data layout of the mesh generated for an FEM using serendipity elements. This layout results in efficient data access and minimizes the memory space allocated for the corresponding mesh.

5.1 Mesh Generation in Two-Dimensional FEMs

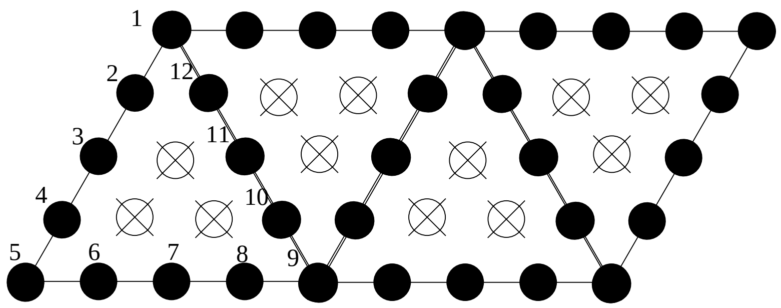
In finite element methods, the problem domain is first discretized into a collection of pre-selected finite elements. Based on discretization, a finite-element mesh of pre-selected elements is generated. The mesh generation method has a strong influence on the quality of the numerical results. In two-dimensional FEMs, the problem domain is usually discretized into triangular or rectangular elements. Some examples are shown in Figure 33.

Different types of elements have different number of pre-selected nodes. After discretization, the geometric properties (e.g., coordinates, cross-section areas, etc.) of each node are generated. Based on these properties, element equations are derived either directly or iteratively. For applications using the iterative approach, the mesh may need to be regenerated for each iteration based on changes in geometric properties.

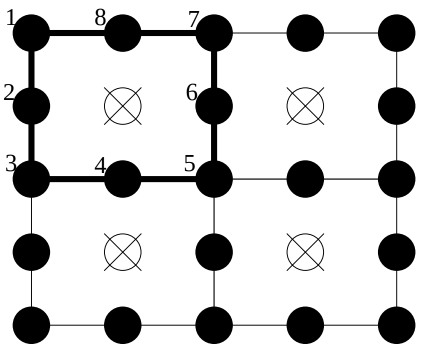
5.2 Efficient Data Layout for the FEMs using Serendipity Elements

In mesh generation of two-dimensional FEMs, internal nodes of the higher-order triangular or rectangular elements can be condensed out at the element level. Condensation is justified as these nodes do not contribute to the inter-element connectivity. Figure 33 (a) shows a mesh with triangular elements, each having 12 boundary nodes and 3 internal nodes. Another mesh with rectangular elements is shown in Figure 33 (b). Here each element has 8 boundary nodes and 1 internal node. We can use the so-called serendipity elements to avoid internal nodes. Serendipity elements are those triangular or rectangular elements which have no interior nodes [26]. Techniques to condense out internal nodes can reduce the size of the element matrices, which in turn are derived from nodal properties. A mesh with serendipity elements can be realized by storing the geometric properties of all nodes in row-major order. However, this straight-forward approach has its problems. First, it wastes memory space. For instance, the FEM shown on Figure 33 (b) uses 8-node rectangular elements, and about $\frac{1}{4}$ of the nodes are not accessed. Second, this approach causes cache pollution when the even rows of the node matrix are accessed, since only half of the nodes are required.

In our work, we have focussed on developing a data layout that addresses the above problems effectively. The goals are to minimize the memory requirements, avoid cache conflict misses, and



(a) 12-node triangular elements



(b) 8-node rectangular elements

Figure 33: Illustration of two FEM meshes using serendipity elements and their node numbering schemes

prevent cache pollution. We suggest an approach in which rows are stored with the same access pattern as a sub-matrix. Various sub-matrices require different indexing schemes based on their corresponding access patterns. Using the technique of separating rows with different access patterns, only the required nodes of each row are stored. In Figure 33 (b), the FEM requires all nodes in the odd rows, and only every other node in the even rows. We stripe the original node matrix row-wise, and alternately store the rows to Matrices A and B respectively (See Figure 34). As depicted in Figure 35, we condense matrix B so that only the nodes to be accessed are stored. Thus, memory requirements are minimized.

Note that both the sub-matrices are needed to access nodes of a single element. In Figure 33 (b), Matrix A is used to access nodes 1, 3, 4, 5, 7, and 8, while Matrix B is used to access nodes 2 and 6. Similarly, the FEM in Figure 33 (a) requires all the nodes in rows 1, 5, 9, ..., $4k + 1$ (k is a non-negative integer). However, it only requires nodes 1,2,5,6,... in rows 2, 6, 10,..., $4k + 2$. Our approach is to alternately store the rows in four sub-matrices.

Table 11 summarizes the sizes and the access patterns of these sub-matrices. We assume an

$n \times m$ node matrix, where n, m are both an integer multiple of 4. In addition to this scheme of separate sub-matrices, we also apply the block data layout. As explained in Section 4, usage of block data layout can further reduce cache conflict misses.

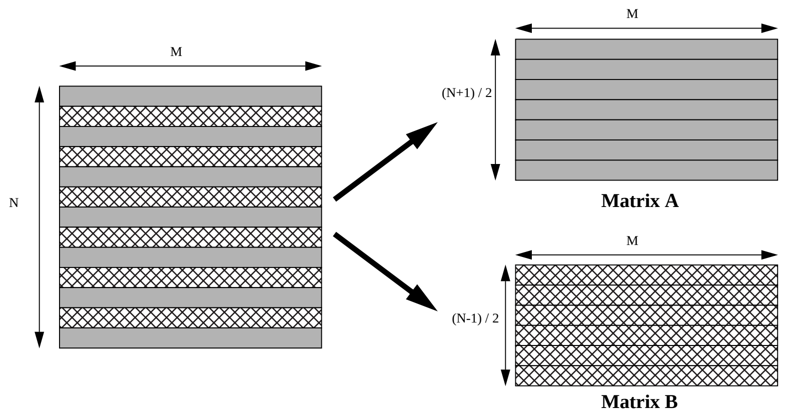


Figure 34: Separation of the odd rows and even rows of a mesh with 12-node rectangular elements

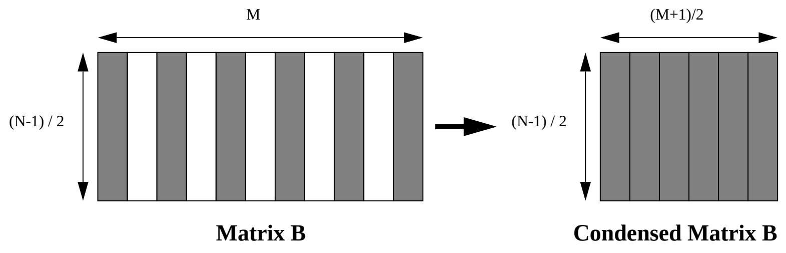


Figure 35: Matrix B is condensed to reduce the overall memory requirement

Using our proposed data layout, both of the sub-matrices are needed to access the nodes of a single element. For instance, in Figure 33 (b), Matrix A is used to access nodes 1, 3, 4, 5, 7, and 8, while Matrix B is used to access nodes 2 and 6.

Similarly, in the other example shown in Figure 33 (a), the FEM requires all of the nodes in rows 1, 5, 9, ..., $4 \times k + 1$ (k is a non-negative integer). However, it only requires nodes 1,2,5,6,... in rows 2, 6, 10,..., $4 \times k + 2$. Using our proposed approach, we alternately store the rows to four sub-matrices. Table 11 summarizes the sizes and the access patterns of these sub-matrices. Note that we assume an $n \times m$ node matrix, where n, m are both an integer multiple of 4. Also, we assume that k is a non-negative integer number.

In addition to the proposed technique to separate the node matrix into sub-matrices, we further apply the block data layout as described in the previous section. The use of block data layout can further reduce the cache conflict misses as explained in Section 4.

5.3 Experimental Results

We have implemented an FEM benchmark using the 8-node rectagular model to examine the efficiency of memory access on an Ultra-SPARC II machine. The matrix in our experiment contains 512 nodes. Table 12 compares the performance of three approaches: a straightforward (without blocking) approach, an approach with blocking, and a scheme that combines separate sub-matrices, blocking, and block data layout. The reported execution times are wall clock times. On UltraSPARC II, we used the gcc compiler with the ”-O3” optimization option. For implementing our proposed scheme, we used block layouts and in-line functions to access each array element. The block size was chosen to be 32 in order to avoid TLB misses.

It is evident from the experimental results in Table 12 that inspite of greater complexity in index computation, our scheme is about 2.2 times faster than the straightforward approach and approximately 1.7 times faster than the blocking based approach.

Table 11: Sub-matrices used for the 12-node triangular FEM

Sub-matrix Name	Sub-matrix Content	matrix size	access pattern
A	Rows $4 \times k + 1$	$(\frac{n}{4})m$	all nodes
B	Rows $4 \times k + 2$	$(\frac{n}{4})(\frac{m}{2})$	1,2,5,6...
C	Rows $4 \times k + 3$	$(\frac{n}{4})(\frac{m}{2})$	1,3,5,7...
D	Rows $4 \times k + 4$	$(\frac{n}{4})(\frac{m}{2})$	1,4,5,8...

Table 12: Memory Access time on UltraSPARC II (400 MHz, 2MByte L2 cache)

FEM Memory Access Benchmark	Memory Access Time (milli-seconds)	Techniques applied
Straightforward Approach	95.313	None
Blocking-Based Approach	74.190	Blocking
Our Proposed Approach	43.718	Separate Sub-matrices Blocking + Block Layout

6 Conclusion

In this report, we have shown techniques for data placement to support efficient memory access for large scale scientific applications.

For large-scale matrix transposition (out-of-core matrix transpose), our algorithm reduced both the number of I/O operations and the index computation time. Our results show that our algorithm reduces the execution time by up to 50%.

For large-scale standard matrix multiplication, our proposed approach using the block data layout has shown significant performance improvement. Our approach reduces cache pollution, conflict cache misses, and TLB misses. Our scheme is up to 15 times faster than naive CBLAS, 2 times faster than blocking based CBLAS, and is 22% faster compared to blocking and copying based CBLAS implementation on UltraSPARC II.

For mesh generation of Finite Element Methods (FEMs), our approach avoids cache pollution. Using blocking and block layout, we achieved further reductions in cache misses and TLB misses. Experimental results indicate speedup by a factor of 2.2 over normal layout and 1.7 over blocking with normal layout.

Our results encourage the use of data placement/data layout techniques to improve the memory access for data-intensive applications. We believe that data layout designs can be further employed in other ERDC applications with complex data access patterns.

7 Acknowlegement

We would like to note the contribution of Mr. Bhaskar Srinivasan for his assistance in preparing the report.

References

[1] A. Aggarwal and J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," Communications of the ACM, Vol. 31, No. 9, pp. 1116-1127, 1988.

[2] M. B. Ari, "On Transposing Large $2^n \times 2^n$ Matrices," IEEE Trans. Computers, Vol. C-27, No. 1, pp. 72-75, 1979.

[3] L. Carter, J. Ferrante and S. F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," Proceedings of IPPS '95, 1995.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High Performance, Reliable Secondary Storage," ACM Computing Surveys, Vol. 26, No. 2, pp. 145-185, June 1994.

[5] A. Choudhary, W. K. Liao, P. Varshney, D. Weiner, R. Linderman and M. Linderman "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers," 12th International Parallel Processing Symposium, Orlando, Florida, 1998.

[6] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur, "Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems," in High Performance Distributed Computing Conference '99, San Diego, CA, August 1999.

[7] T. H. Cormen, T. Sundquist, and L. F. Wisniewski, "Asymptotically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems," Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, 1994.

[8] J. C. Curlander and R. N. McDonough, Synthetic Aperture Radar-Systems and Signal Processing, Wiley, 1991.

[9] L. G. Delcaro and G. L. Sicuranza, "A Method on Transposing Externally Stored Matrices," IEEE Trans. on Computers, Vol. C-23, No. 9, pp. 801-803, 1974.

[10] M. Kallahalla and P. Varman, "Optimal Read-Once Parallel Disk Scheduling," Proc. ACM Workshop on I/O in Parallel and Distributed Systems, April 1999.

[11] M. Kallahalla and P. Varman, "An Improved Parallel Prefetching Algorithm," Proc. of Intl. Conference on High Performance Computing, Dec. 1998.

[12] D. E. Dudgen and R. M. Mersereau, Multidimensional Signal Processing, Prentice-Hall, 1984.

[13] J. O. Eklundh, "A Fast Computer Method for Matrix Transposing," IEEE Transactions on Computers, Vol. 20, Number 7, pp. 801-803, 1972.

[14] R. W. Floyd, “Permuting Information in Idealized Two-level Storage, ” Complexity of Computer Computations, pp. 105-109, Plenum, 1972.

[15] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, “Efficient Transposition Algorithms for Large Matrices”, Supercomputing, 1993.

[16] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, The High Performance Fortran Handbook, The MIT Press, 1994.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1994.

[18] W. Liao, A. Choudhary, D. Weiner, and Pramod Varshney, “Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes, IPPS/SPDP, San Juan, Puerto Rico, 1999.

[19] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, “Efficient Algorithms for Block-Cyclic Redistribution of Arrays,” 24:298-330, Algorithmica, 1999.

[20] http://www.darpa.mil/ito/research/dis/index.html, 1999.

[21] H. Park, J. Suh, V. K. Prasanna, and M. Ung, ”Parallel Implementation of 2D FFT on High Performance Computing Platforms,” DoD HPC User’s Conference ’98, Houston, Texas, June 1998.

[22] H. K. Ramapriyan, “A Generalization of Eklundh’s Algorithm for Transposing Large Matrices,” IEEE Trans. on Computers, Vol. C-24, No. 12, pp. 1221-1226, 1975.

[23] S. Ranka and S. Sahni, Hypercube Algorithms for Image Processing and Pattern Recognition, Springer-Verlag, New York, New York, 1990.

[24] J. Suh and V. K. Prasanna, “Portable Implementation of Real Time Signal Processing Benchmarks on HPC Platforms,” International Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems ’98, Umea, Sweden, June 1998.

[25] J. S. Vitter and E. A. M. Shriver, “Algorithms for Parallel Memory I: Two-level Memories,” Algorithmica, Vo. 12 No. 2-3, pp. 110-147, 1994.

[26] J. N. Reddy, ”An Introduction to the Finite Element Method,” published by McGraw-Hill Inc., 1984.